

Review

Operating Systems

Processes



Maria Hybinette, UGA

- Operating System Fundamentals
 - What is an OS?
 - What does it do?
 - How and when is it invoked?
- Structures
 - Monolithic
 - Layered
 - Microkernels
 - Virtual Machines
 - Modular

Maria Hybinette, UGA

Chapter 3: Processes: Outline

- Process Concept: views of a process
- Process Basics Scheduling Principles
- Operations on Processes
 - Life of a process: from birth to death ...

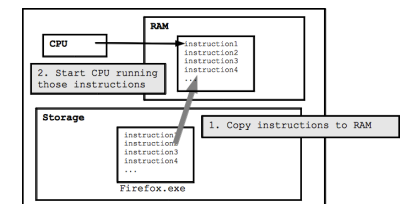


- Cooperating Processes
 - Inter process Communication – local and remote
 - Mailboxes
 - Shared Memory
 - Sockets
 - Pipes
 - Files

Maria Hybinette, UGA

What is a Process?

- A program in execution
- An activity
- A *running* program.
 - Basic unit of work on a computer, a job, or a task.
 - A container of instructions with some resources:
 - e.g. CPU time (CPU carries out the instructions), memory, files, I/O devices to accomplish its task



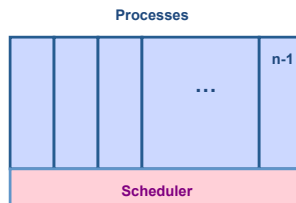
Examples: compilation process, word processing process, scheduler (sched, swapper) process or daemon processes: ftpd, http

Maria Hybinette, UGA

What are Processes?

System View:

- **Multiple** processes:
 - Several distinct processes can execute the SAME program
- Time sharing systems run several processes by multiplexing between them (scheduler manages the multiplexing).
- ALL “runnables” including the OS are organized into a number of “sequential processes”

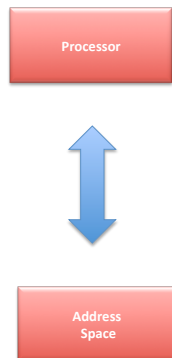


Maria Hybinette, UGA

Process Definition

A **process** is a ‘program in execution’, a **sequential execution** characterized by trace.

It has a **context** (the information or data) and this ‘context’ is maintained as the process progresses through the system.



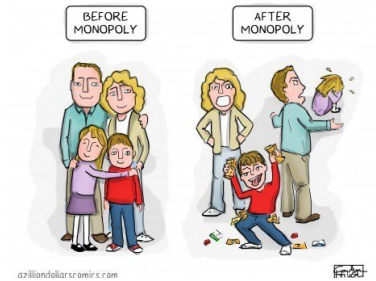
Maria Hybinette, UGA



Virtualization

User View:

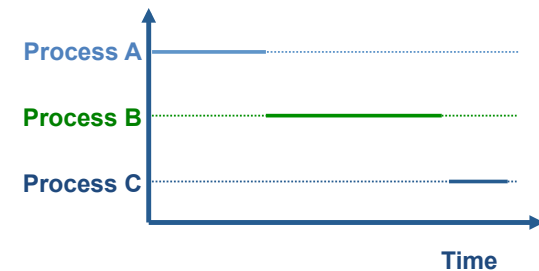
- Processor → Virtual Processor
 - Illusion of monopolizing the processor
- Memory → Virtual Memory
 - Address space! Really – more than just memory.
 - Illusion of monopolizing address space; or access of all memory of the system and open files and other resources.



Maria Hybinette, UGA



Activity of a Process

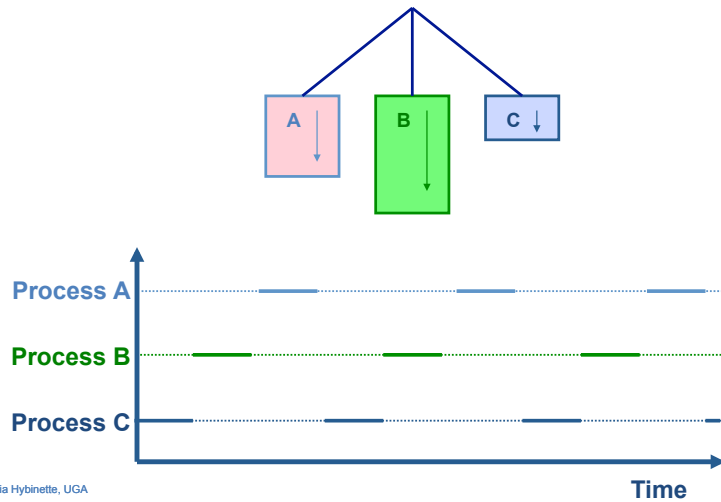


Multiprogramming? How?

- **Solution:** Provide a programming counter.
- One processor (CPU). Give an illusion that each process has its own processor.

Maria Hybinette, UGA

Activity of a Process: Time Sharing



Maria Hybinette, UGA

Enabling “Processing”

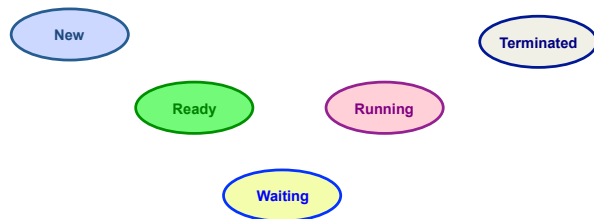
Investigate: What Does the Process Do?

- **Begin:** It is created!
 - Runs
 - Does not run (but ready to run)
 - Runs
 - Does not run (but ready to run)
 -
- **End:** Terminates

Maria Hybinette, UGA

‘States’ of a Process

- As a process executes, it changes *state*
 - **New:** The process is being created.
 - **Running:** Instructions are being executed.
 - **Ready:** The process is waiting to be assigned to a processor (CPU).
 - **Terminated:** The process has finished execution.
 - **Waiting:** The process is waiting for some event to occur.



Maria Hybinette, UGA

State Transitions

- A process may change state as a result:
 - Program action (system call)
 - OS action (scheduling decision)
 - External action (interrupts)



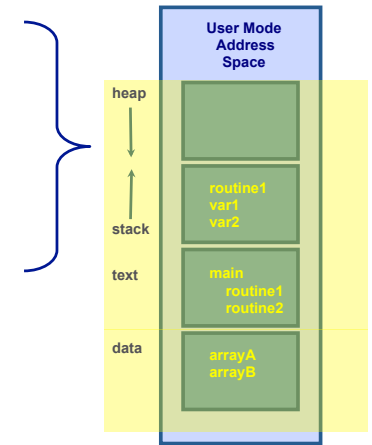
Maria Hybinette, UGA

OS Designer's Questions?

- How is process state represented?
 - What information is needed to represent a process?
- How are processes **selected** to transition between states?
- What mechanism is needed for a process to run on the CPU?

What Makes up a Process?

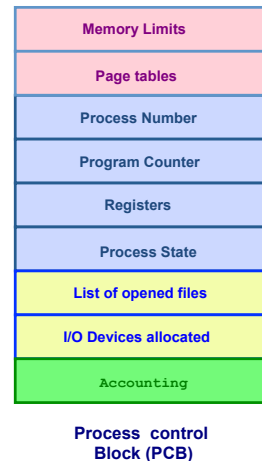
- User resources/OS Resources:
- Program code (text)
 - global variables
 - heap (dynamically allocated memory)
 - Data
 - function parameters
 - return addresses
 - local variables and functions
 - OS Resources, environment
 - open files, sockets
 - Credential for security
 - Registers
 - program counter, stack pointer



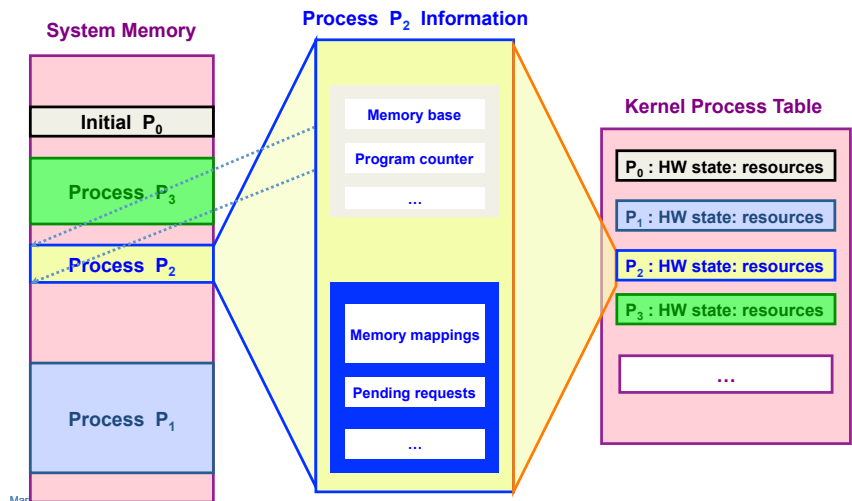
address space are the shared resources of a(l) thread(s) in a program

What is needed to keep track of a Process?

- **Memory information:**
 - Pointer to memory segments needed to run a process, i.e., pointers to the address space -- text, data, stack segments.
- **Process management information:**
 - Process state, ID
 - Content of registers:
 - Program counter, stack pointer, process state, priority, process ID, CPU time used
- **File management & I/O information:**
 - Working directory, file descriptors open, I/O devices allocated
- **Accounting:** amount of CPU used.



Process Representation



OS View: Process Control Block (PCB)

- How does an OS keep track of the state of a process?
 - Keep track of 'some information' in a structure.
 - Example: In Linux a process' information is kept in a structure called `struct task_struct` declared in `#include linux/sched.h`
 - What is in the structure?
- Where is it defined:
 - not in `/usr/include/linux` - only user level code
 - `/usr/src/kernels/2.6.32-642.3.1.el6.x86_64/include/linux`
 - » (on nike).

```

struct task_struct
  pid_t pid;          /* process identifier */
  long state;        /* state for the process */
  unsigned int time_slice /* scheduling information */
  struct mm_struct *mm /* address space of this process */
    
```

State in Linux

```

volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */

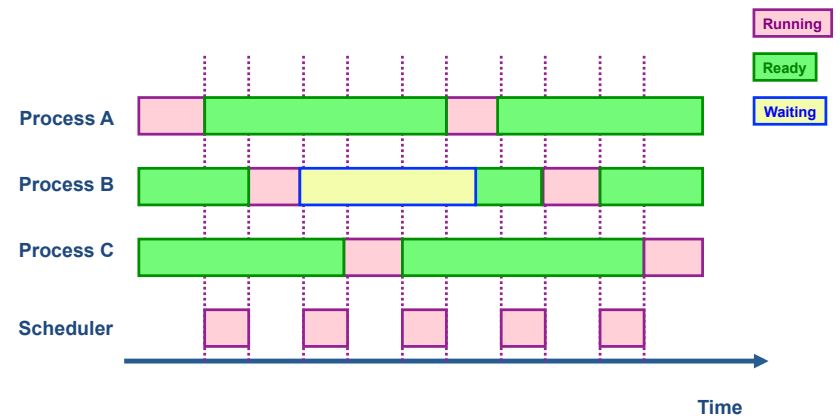
#define TASK_RUNNING      0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_ZOMBIE      4
#define TASK_STOPPED     8
#define TASK_EXCLUSIVE   32
    
```

- traditionally 'zombies' are child processes of parents that have not processed a `wait()` instruction.
- Note: processes that have been 'adopted' by `init` are not zombies (these are children of parents that terminates before the child). `init` automatically calls `wait()` on these children when they terminate.
- this is true in LINUX.
- What to do: 1) Kill the parent 2) Fix the parent (make it issue a `wait()`) 2) Don't care

Process Table in Microkernel (e.g., MINIX)

- Microkernel design - process table functionality (monolithic) partitioned into four tables:
 - Kernel management (kernel/proc.h)
 - Memory management (VM server `vm/vmproc.h`)
 - Memory part of `fork`, `exit` etc calls
 - Used/unused part of memory
 - File management (FS) (FS server `fs/fproc.h`)
 - Process management (PM server `pm/mproc.h`)

Running Processes



Why is Scheduling important?

- Goals:
 - Maximize the ‘usage’ of the computer system
 - Maximize CPU usage (utilization)
 - Maximize I/O device usage
 - Meet as many task deadlines as possible (maximize throughput).

HERE

Maria Hybinette, UGA

Process Characteristics

- Processes can be described as either:
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts.
 - **CPU-bound process** – spends more time doing computations; few very long CPU bursts.

Maria Hybinette, UGA

Scheduling

- **Approach:** Divide up scheduling into task levels:
 - Select process who gets the CPU (from main memory).
 - Admit processes into memory
 - Sub problem: How?
- **Short-term scheduler (CPU scheduler):**
 - selects which process should be executed next and allocates CPU.
 - invoked frequently (ms) ⇒ (must be fast).
- **Long-term scheduler (look at first):**
 - selects which processes should be brought into the memory (and into the ready state)
 - invoked infrequently (seconds, minutes)
 - controls the *degree of multiprogramming*.

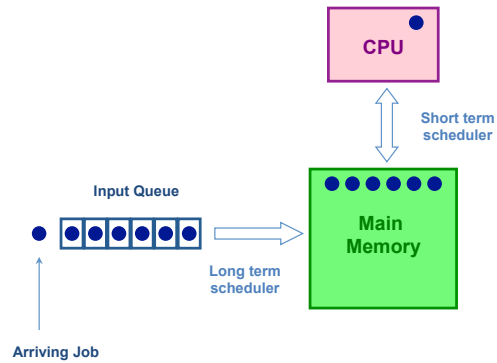
Maria Hybinette, UGA

Observations

- If all processes are I/O bound, the ready queue will almost always be empty (little scheduling)
- If all processes are CPU bound the I/O devices are underutilized
- **Approach (long term scheduler):** ‘Admit’ a good mix of CPU bound and I/O bound processes.

Maria Hybinette, UGA

Big Picture (so far)



Maria Hybinette, UGA

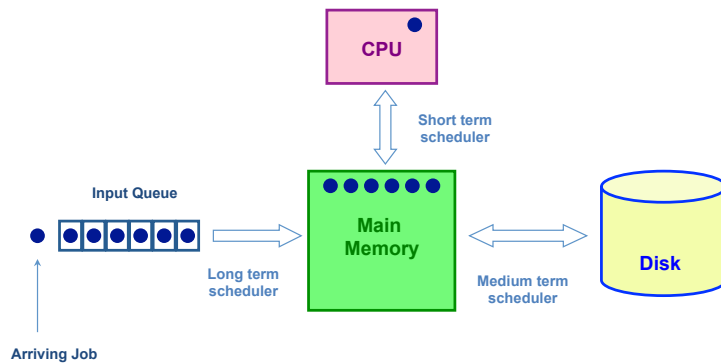
Exhaust Memory?

- **Problem:** What happens when the number of processes is so large that there is not enough room for all of them in memory?
- **Solution:** Medium-level scheduler:
 - Introduce another level of scheduling that removes processes from memory; at some later time, the process can be reintroduced into memory and its execution can be continued where it left off
 - Also affect degree of multi-programming.

Maria Hybinette, UGA

Which processes should be selected?

- Processor (CPU) is faster than I/O so all processes could be waiting for I/O
 - Swap these processes to disk to free up more memory
- Blocked state becomes suspend state when swapped to disk
 - Two new states
 - waiting, suspend
 - Ready, suspend

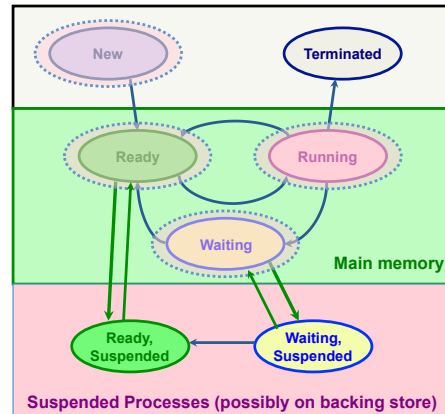


Maria Hybinette, UGA

Maria Hybinette, UGA

Suspending a Process

- Which to suspend?
- Others?



Maria Hybinette, UGA

Possible Scheduling Criteria

- How long since process was swapped in or out?
- How much CPU time has the process had recently?
- How big is the process (small ones do not get in the way)?
- How important is the process (high priority)?

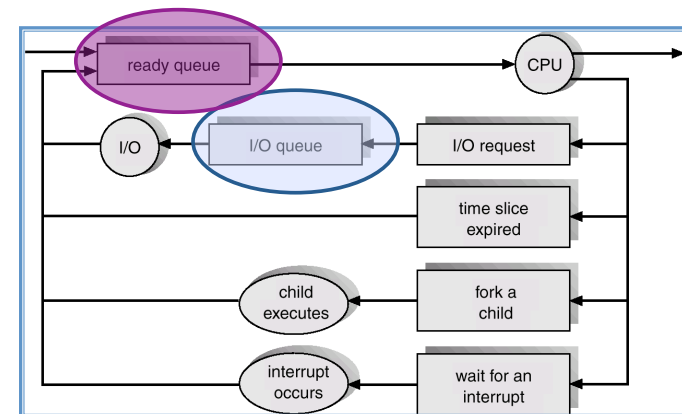
Maria Hybinette, UGA

OS Implementation: Process Scheduling Queues

- **Job queue** – set of all processes in the system.
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute on CPU
- **Device queues** – set of processes waiting for an I/O device.
- Process migration between the various queues.

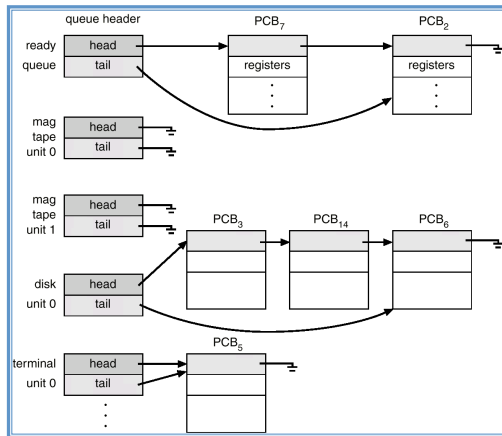
Maria Hybinette, UGA

Representation of Process Scheduling



Maria Hybinette, UGA

Ready Queue, I/O Device Queues



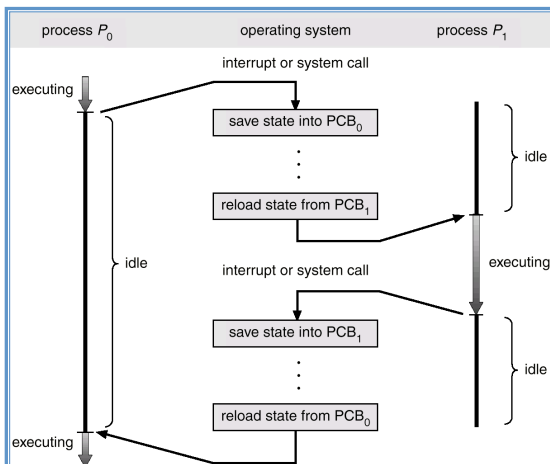
Maria Hybinette, UGA

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.

Maria Hybinette, UGA

CPU Context Switches



Maria Hybinette, UGA

Process Creation

- **Process Cycle:** Parents create children; results in a (inverse) tree of processes.
 - Forms an ancestral hierarchy
- **Address space models:**
 - Child duplicate of parent.
 - Child has a program loaded into it.
- **Execution models:**
 - Parent and children execute concurrently.
 - Parent waits until children terminate.
- **Examples**

Maria Hybinette, UGA

Continuing the Boot Sequence...

- After loading in the Kernel and it does a number of system checks it creates a number of 'dummy processes' -- processes that cannot be killed -- to handle system tasks.
- A common approach (UNIX) is to create processes in a tree process structure

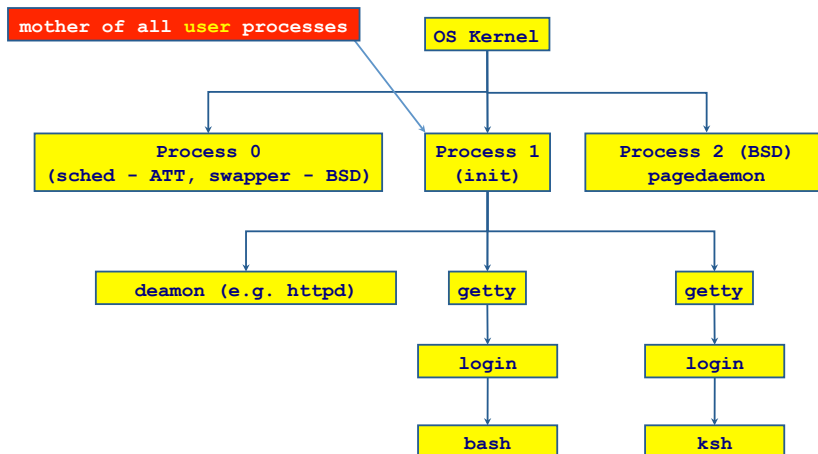
Process Life Cycle: UNIX (cont)

- PID 0 is *usually* the `sched` process (often called `swapper` -- which handles memory/page mapping of processes).
 - is a `system process` -- **** it is part of the kernel ****
 - the grandmother of *all* processes).
- `init` - Mother of all `user processes`, `init` is started at boot time (at *end of the boot strap* procedure) and is responsible for starting other processes
 - It is a user process (not a system process that runs within the kernel like `swapper`) with PID 1 (but runs with root privileges)
 - `init` uses file `inittab` and directory `/etc/rc?.d`
 - brings the user to a certain specified state (e.g., multiuser mode)
 - Daemons (background process):
 - [http://en.wikipedia.org/wiki/Daemon_\(computing\)](http://en.wikipedia.org/wiki/Daemon_(computing))
- `getty` - login process that manages login sessions

Maria Hybinette, UGA

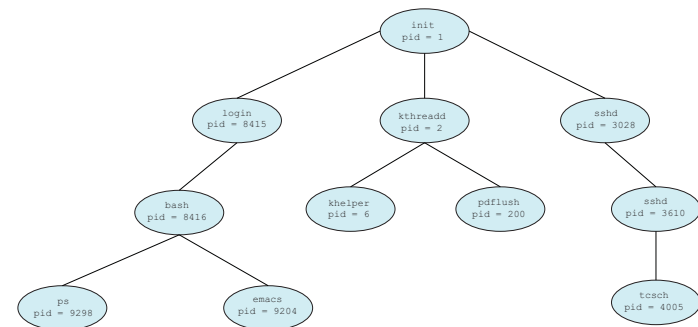
Maria Hybinette, UGA

Processes Tree on a typical UNIX System



Maria Hybinette, UGA

Linux Specific Process Tree



Maria Hybinette, UGA

Other Systems

HP-UX 10.20

UID	PID	PPID	C	STIME	TTY	TIME	COMMAND
root	0	0	0	Apr 20 ?		0:17	swapper
root	1	0	0	Apr 20 ?		0:00	init
root	2	0	0	Apr 20 ?		1:02	vhand

Linux RedHat 6.0:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	09:59 ?		00:00:07	init
root	2	1	0	09:59 ?		00:00:00	[kflushd]
root	3	1	0	09:59 ?		00:00:00	[kpiod]
root	4	1	0	09:59 ?		00:00:00	[kswapd]
root	5	1	0	10:00 ?		00:00:00	[mdrecoveryd]

Solaris:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	0	0	0	Apr 19 ?		0:00	sched
root	1	0	0	Apr 19 ?		0:22	/etc/init -
root	2	0	0	Apr 19 ?		0:00	pageout

* sched - dummy process which provides swapping services
 * pageout - dummy process which provides virtual memory (paging) services

Maria Hybinette, UGA

Running Processes

- Print out status information of various processes in the system: `ps -axj (BSD)` , `ps -efjc (SVR4)`
- Daemons (background processes) with root privileges, no controlling terminal, parent process is `init`

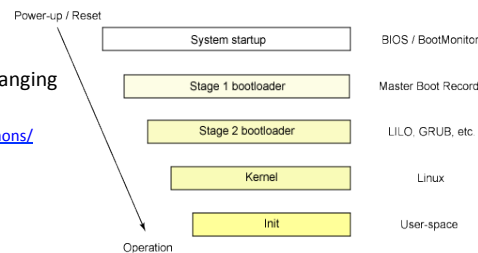
```
{atlas:maria} ps -efjc | sort -k 2 -n | less
{nike:maria} ps -ajx | sort -n -k 2 | less
```

UID	PID	PPID	PGID	SID	CLS	PRI	STIME	TTY	TIME	CMD
root	0	0	0	0	SYS	96	Mar 03 ?		0:01	sched
root	1	0	0	0	TS	59	Mar 03 ?		1:13	/etc/init -r
root	2	0	0	0	SYS	98	Mar 03 ?		0:00	pageout
root	3	0	0	0	SYS	60	Mar 03 ?	4786:00		fsflush
root	61	1	61	61	TS	59	Mar 03 ?		0:00	/usr/lib/sysevent/syseventd
root	64	1	64	64	TS	59	Mar 03 ?		0:08	devfsadmd
root	73	1	73	73	TS	59	Mar 03 ?		30:29	/usr/lib/picl/picld
root	256	1	256	256	TS	59	Mar 03 ?		2:56	/usr/sbin/rpcbind
root	259	1	259	259	TS	59	Mar 03 ?		2:05	/usr/sbin/keyserv
root	284	1	284	284	TS	59	Mar 03 ?		0:38	/usr/sbin/inetd -s
daemon	300	1	300	300	TS	59	Mar 03 ?		0:02	/usr/lib/nfs/statd
root	302	1	302	302	TS	59	Mar 03 ?		0:05	/usr/lib/nfs/lockd
root	308	1	308	308	TS	59	Mar 03 ?		377:42	/usr/lib/autofs/automountd
root	319	1	319	319	TS	59	Mar 03 ?		6:33	/usr/sbin/syslogd

Maria Hybinette, UGA

Linux Processes/Daemons

- Linux processes (`ps -ef`)
 - `ps tree -a 1` (see the hierarchy of processes starting at pid 1).
 - `lsdf` (list of open files)
 - `htop`, `atop`, `top` (process viewer, interactive version of `ps`)
- Read:
 - <http://www.ibm.com/developerworks/library/l-linuxboot/index.html>
- Linux Daemons:
 - `watchdog`
 - Timer prevent system from hanging
 - `ksoftirqd`
 - <http://www.sorgonet.com/linux/linuxdaemons/>



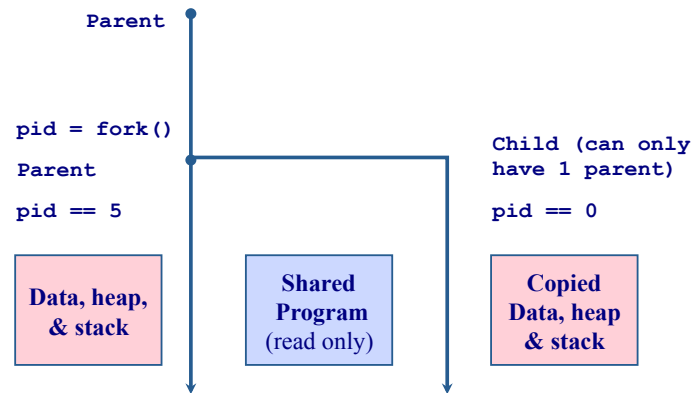
Maria Hybinette, UGA

Process Creation: Execution & Address Space in UNIX

- In UNIX process `fork()` - `exec()` mechanisms handles process creation and its behavior:
 - `fork()` creates an exact copy of itself (the parent) and the new process is called the child process
 - `exec()` system call places the image of a new program over the newly copied program of the parent

Maria Hybinette, UGA

fork() a child



Maria Hybinette, UGA

Example: parent-child.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int i;
    pid_t pid;
    pid = fork();
    if( pid > 0 )
    {
        /* parent */
        for( i = 0; i < 1000; i++ )
            printf( "\tPARENT %d\n", i );
    }
    else
    {
        /* child */
        for( i = 0; i < 1000; i++ )
            printf( "\tCHILD %d\n", i );
    }
}
```

```
{saffron} parent-child
PARENT 0
PARENT 1
PARENT 2
CHILD 0
CHILD 1
PARENT 3
PARENT 4
CHILD 2
.
```

Maria Hybinette, UGA

Things to Note

- `i` is copied between parent and child
- The switching between parent and child depends on many factors:
 - Machine load, system process scheduling, ...
- I/O buffering effects the output shown
 - Output interleaving is *non-deterministic*
 - Cannot determine output by looking at code

Maria Hybinette, UGA

Process Creation: Windows

- Processes created via 10 params `CreateProcess()`
- Child process *requires* loading a specific program into the address space.

```
BOOL WINAPI CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation );
```

Maria Hybinette, UGA

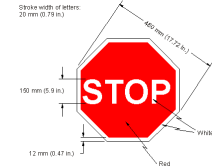
Process Termination

- Process executes last statement and asks the operating system to delete it by using the `exit()` system call.
 - Output data from child to parent (via wait).
 - Process' resources are deallocated by operating system. May not be immediate.
- Parent may terminate execution of **children processes** (abort).
 - Child has exceeded allocated resources.
 - Task assigned to child is no longer required.
 - Parent is exiting.
 - Some Operating system does not allow child to continue if its parent terminates.
 - Cascading termination (initiated by system to kill of children of parents that exited).
 - If a parents terminates - children are adopted by `init()` - so they still have a parent to collect their status and statistics

Maria Hybinette, UGA

Processes

- **Independent** process cannot affect or be affected by the execution of another process.
- **Cooperating** process can affect or be affected by the execution of another process
 - **Advantages** of process cooperation
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience
 - **Requirement:** Inter-process communication (IPC) mechanism.



Maria Hybinette, UGA

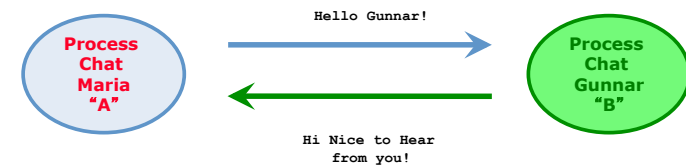
Two Communicating Processes



- Concept that we want to implement
- We can do this already – via pipes
 - Problem:

Maria Hybinette, UGA

Two Communicating Processes



- Concept that we want to implement
- We can do this already – via pipes

Maria Hybinette, UGA

On the path to communication ...

- **Want:** A communicating processes
- Have so far: Forking – to create processes
- Problem:
 - After fork() is called we end up with two **independent** processes.
 - Separate Address Spaces
- Solution? How do we communicate?

How do we communicate?

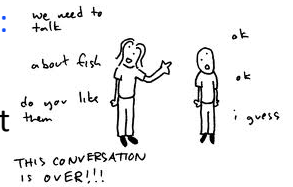
Local Machine:

- Files (done) write to a file read from file
- Pipes (done)
- Signals (covered in system's progms)
- ...



Remote Machines: 2 Primary Paradigms:

- (Distributed) Shared Memory
- Messages (this paradigm also extends to Remote Machines)

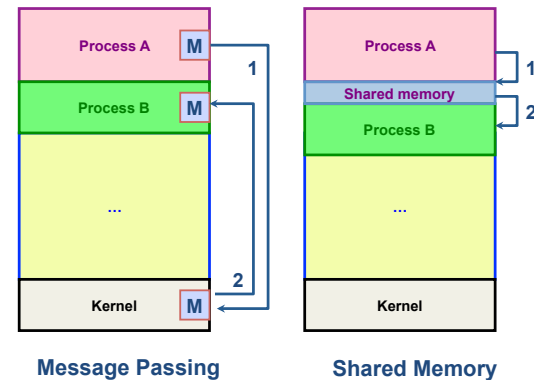


–[Same machine, Remote Machines, RPC].

Communication Models

- Shared memory model
 - Share memory region for communication
 - Read and write data to **shared region**
 - Typically Requires synchronization (e.g., locks)
 - Faster than message passing
 - Setup time
- Message Passing model
 - Communication via exchanging messages

Communication Models



Communication Implementations

- Within a single computer

- Pipes (done)
 - Unamed: only persist as long as process lives
 - Named Pipes (FIFO)- looks like a file (`mkfifo filename, attach, open, close, read, write`)
 - http://developers.sun.com/solaris/articles/named_pipes.html
- Message Passing (message Queues, next HW)
- Shared Memory (next HW)

- Distributed System (remote computers, connected via cable, air e.g., WiFi) - Later

- TCP/IP sockets
- Remote Procedure Calls (next project)
- Remote Method Invocations (RMI, maybe project)
- Message passing libraries: MPI, PVM

Maria Hybinette, UGA




Message Passing Systems

- NO shared state

- Communicate across address spaces and protection
- Agreed protocol

- Generic API

- `send(dest, &msg)`
- `recv(src, &msg)`

- What is the dest and src?

- A pid : e.g., with signals
- A File: e.g., pipe
- Port, network address, queue
- Unspecified source (any source, any message)

Maria Hybinette, UGA



- Explicitly specify dest and src process by an identifier

- Multiple buffers:

- Receiver
 - If it has multiple senders (then need to search through a 'buffer(s)' to get a specific sender)
- Sender

- What is the dest and src IDs?

- A PID : e.g., signals.
- A File: e.g., pipe
- A Port, network address,
- Unspecified source (any source, any message)

Maria Hybinette, UGA

- Message Queues:

- dest and src are (unique) queues (end point)

- Uses a unique **shared queue**, allows many to many communication :

- messages sorted FIFO
- messages are stored as a sequence of bytes
- get a message queue identifier (can create queue)

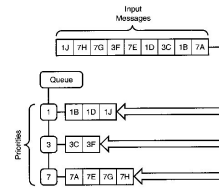

```
int queue_id = msgget ( key, flags )
```

- sending messages:

- `msgsnd(queue_id, &buffer, size, flags)`

- receiving messages (type is priority):

- `msgrcv(queue_id, &buffer, size, type, flags)`



Maria Hybinette, UGA

Demo

- Kirk.c
- Spock.c

Mailboxes vs Pipes

- Same machine: Are there any differences between a mailbox and a pipe?
 - Message types
 - mailboxes may have messages of different data types
 - pipes do not have different types
- Buffer
 - Pipes: Messages stored in contiguous bytes
 - Mailbox – linked list of messages of different types
- Number of processes
 - Typically 2 for pipes (one sender & one receiver)
 - Many processes typically use a mailbox (understood paradigm)

Maria Hybinette, UGA

Courtesy: Beej

Maria Hybinette, UGA

Shared Memory

- Efficient and fast way for processes to communicate
 - After setting up a shared memory segment
- Process: Create, Attach, Populate, Detach
 - create a shared memory segment
 - `shmid = shmget(key, size, flags)`
 - attach a sms to a data space:
 - `shmat(shmid, *shmaddr, flags)`
 - Populate or Read/Write (with regular instructions)
 - detach (close) a shared segment:
 - `shmdt(*shmaddr)`– synchronized.

if more than one process can access segment, an outside protocol or mechanism (like semaphores) should enforce consistency and avoid collisions

Simple Example: `shm_server.c` and `shm_client.c`

Maria Hybinette, UGA

```
shm_server.c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ 27

main()
{
    int shmid;
    key_t key;
    char c, *shm, *s;

    key = 5678; /* selected key */

    /* Create the segment.*/
    if ((shmid = shmget(key,SHMSZ,IPC_CREAT | 0666)) <
        0)
        perror("shmget"); exit(1);

    /* Attach the segment to our data space.*/
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
    {
        perror("shmat"); exit(1);
    }

    /* Populate/Write: put some things into memory */
    for (s = shm, c = 'a'; c <= 'z'; c++) *s++ = c;
    *s = NULL;

    /* Read: wait until first character changes to '*'
    */
    while (*shm != '*') sleep(1);

    exit(0);
}
```

```
shm_client
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ 27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    key = 5678; /* selected key by server */

    /* Locate the segment. */
    if ((shmid = shmget(key,SHMSZ,0666)) < 0)
    {
        perror("shmget"); exit(1);
    }

    /* Attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) \
    {
        perror("shmat"); exit(1);
    }

    /* Read what the server put in the memory. */
    for (s = shm; *s != NULL; s++) putchar(*s);
    putchar('\n');

    /* Write/Synchronize change the first character in
    segment to '*' */
    *shm = '*';

    exit(0);
}
```


Synchronous/Asynchronous Commands

- Synchronous – e.g., blocking (wait until command is complete, e.g., block read or receive).
 - Synchronous Receive:
 - receiver process waits until message is copied into user level buffer
- Asynchronous – e.g., non-blocking (don't wait)
 - Asynchronous Receive
 - Receiver process issues a receive operation and then carries on with task
 - Polling – comes back to see if receive as completed
 - Interrupt – OS issues interrupt when receive has completed

Maria Hybinette, UGA

Buffering

- Queue of messages attached to link:
 - Zero capacity
 - 0 message - link cannot have any messages waiting
 - Sender must wait for receiver (rendezvous)
 - Bounded capacity
 - n messages - finite capacity of n messages
 - Sender must wait if link is full
 - Unbounded capacity
 - infinite messages -
 - Sender never waits

Maria Hybinette, UGA

Synchronous: OS view vs Programming Languages

- OS View:
 - synchronous send ⇒ sender blocks until message has been copied from application buffers to the kernel
 - Asynchronous send ⇒ sender continues processing after notifying OS of the buffer in which the message is stored; have to be careful to not overwrite buffer until it is safe to do so
- PL view:
 - synchronous send ⇒ sender blocks until message has been received by the receiver
 - asynchronous send ⇒ sender carries on with other tasks after sending message

Maria Hybinette, UGA

Remote Machine Communication

- Socket communication (do on your own, bonus available, with tutorial and code snippets): Interested?
- Remote Procedure Calls RPC (right now)
- Remote Method Invocation (next week)

HW 3 – later will be a one week HW RPC & RMI

Maria Hybinette, UGA