CS-446/646

Processes

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Process Abstraction

Process

The OS *Abstraction* for CPU / Execution

- The unit of Execution
- The unit of *Scheduling*
- The *"Dynamic Execution Context"* of a Program
- Also called a "*Job*" or a "*Task*"

A *Process* is a program *in execution*

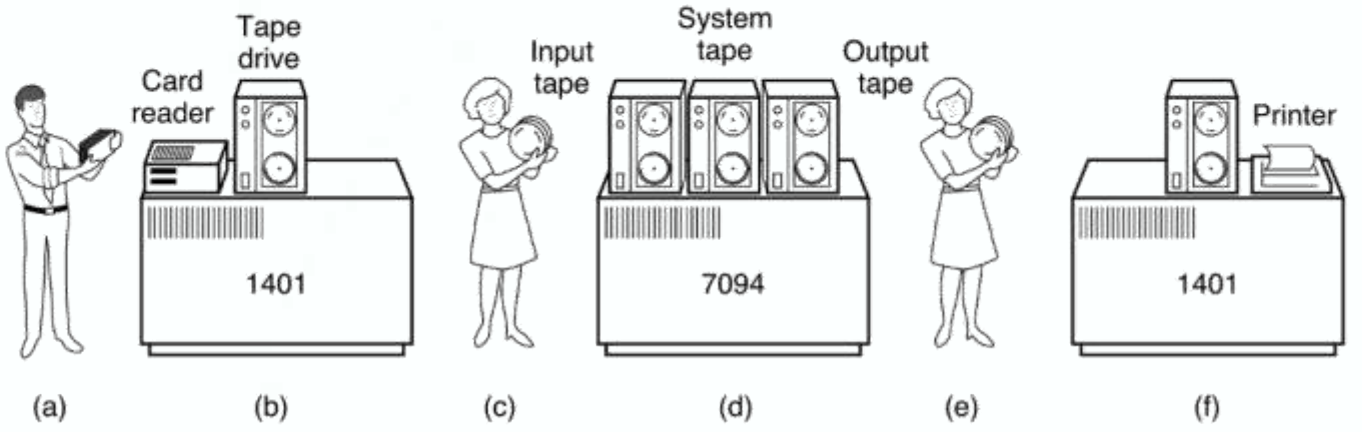
- Defines the sequential, instruction-at-a-time execution of a Program
- Programs are static entities with the *potential* for execution

Program ≠ *Process*

- Program: Static code + static data
- *Process*: Dynamic instantiation of code + data + more

Process Management

Uniprogramming - Simple Process Management: One-at-a-time



circa 1960s

A *Process* runs from start to full completion

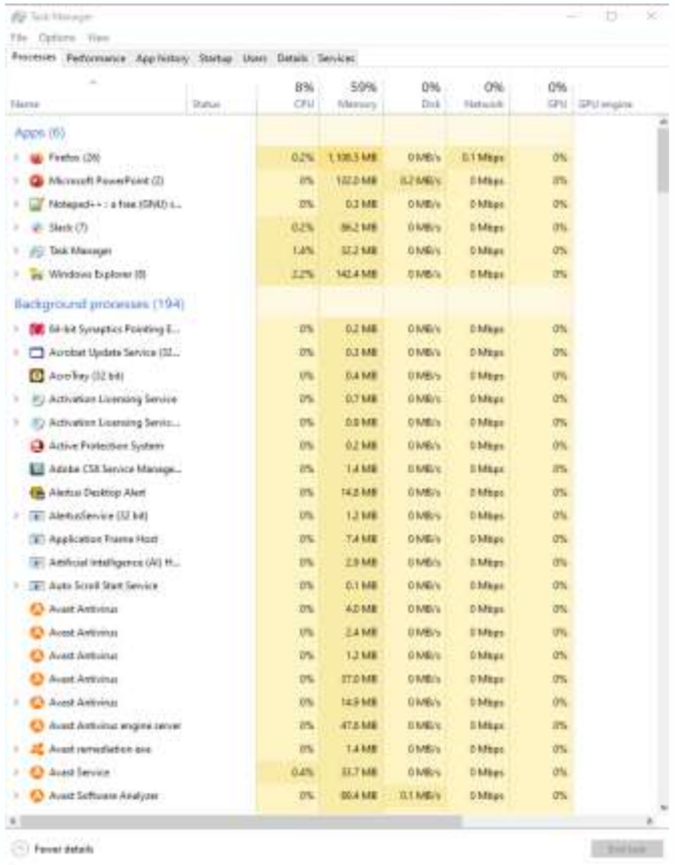
- What the early batch operating system does
- Load a job from disk (tape) into memory, execute it, unload the job
- Problem: Low utilization of Hardware
  - An I/O-intensive *Process* would spend most of its time waiting for punched cards to be read
  - CPU time is wasted
  - Computers were very expensive back then

Process Management

Multiprogramming – Multiple *Process* Management: Multiple at-the-same-time

Modern OSs run multiple processes *Concurrently*

- Example:
  - **`gcc file_A.c`** – compiler running on file A
  - **`gcc file_B.c`** – compiler running on file B
  - **`vim`** – text editor
  - **`firefox`** – web browser
- Note:
  - Multiple **`firefox`** windows:
    - Implemented as a single *Process* (recent versions actually implement splitting into different *Processes*, but still the overall collection serves to manage all open window instances)



Windows 10 Task Manager

Process Management

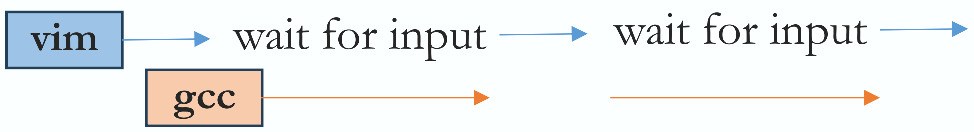
Multiprogramming – Multiple *Process* Management : Multiple at-the-same-time

- Multiple *Processes* loaded in Memory and available to run
- If a process is *Blocked* in I/O, select another *Process* to run on CPU
- Different Hardware components utilized by different *Tasks* at the same time

Advantages: Increase utilization and overall speed

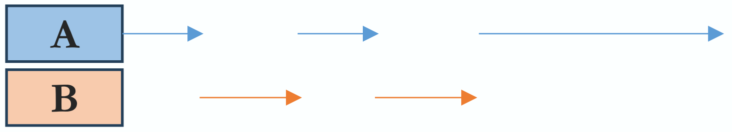
- Higher throughput, lower latency

- Multiple *Processes* can increase CPU utilization
  - Overlap one *Process*' computation with another's wait



- Multiple *Processes* can reduce *Latency*
  - Running A then B requires 100 sec for B to complete

  - Running A and B *Concurrently* makes B finish faster (and more responsive)



Note: A will however run slower than if it had whole machine to itself

Process Components

A *Process* contains all *State* for a Program in execution

- An *Address Space*
- The *Code* for the executing Program
- The *Data* for the executing Program
- An *Execution Stack* encapsulating the state of procedure calls
- The *Program Counter* (PC) indicating the next *Instruction*
- A set of *General-Purpose Registers* with current values
- A set of Operating System *Resources*
  - Open *Files*, Network Connections, etc.

Kernel & *Processes*

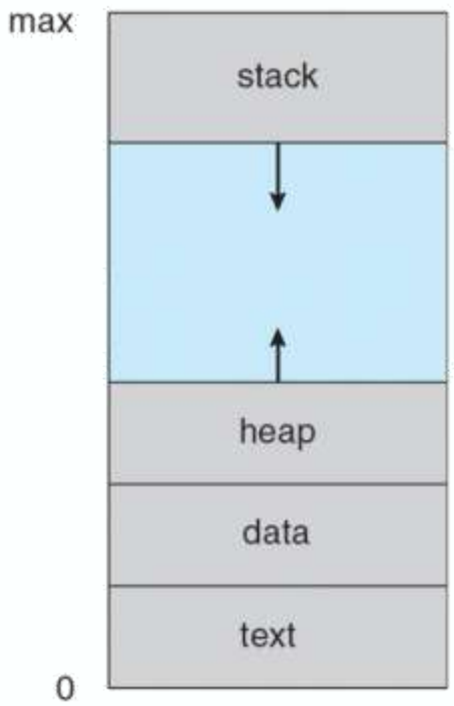
From the ***Process'*** viewpoint

Each process has own view of the machine

- Its own *Virtual Address Space*
- Its own *Virtual CPU*
- Its own (virtually-exclusive) open *Files*

`*(char *)0xc000` means a different thing in *Process1* & *Process2*

Note: Above is C code.



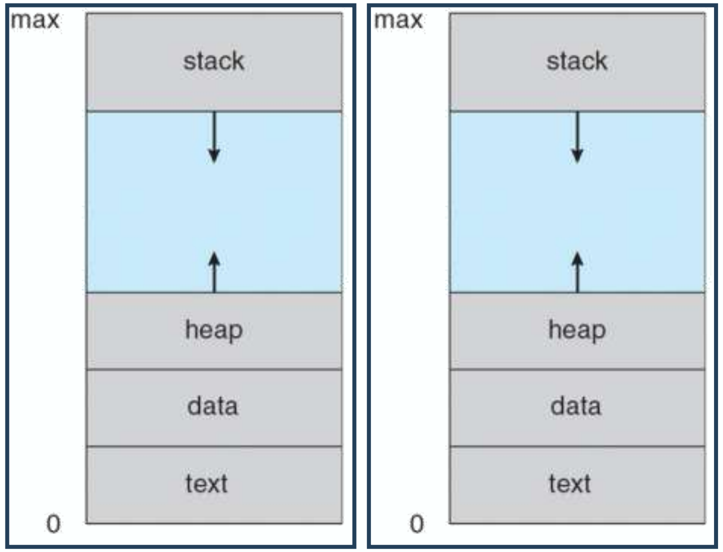
- Simplifies programming model
  - `gcc` does not care that `firefox` is running

Process Address Space

Address Space: All memory a *Process* **can** (potentially) address

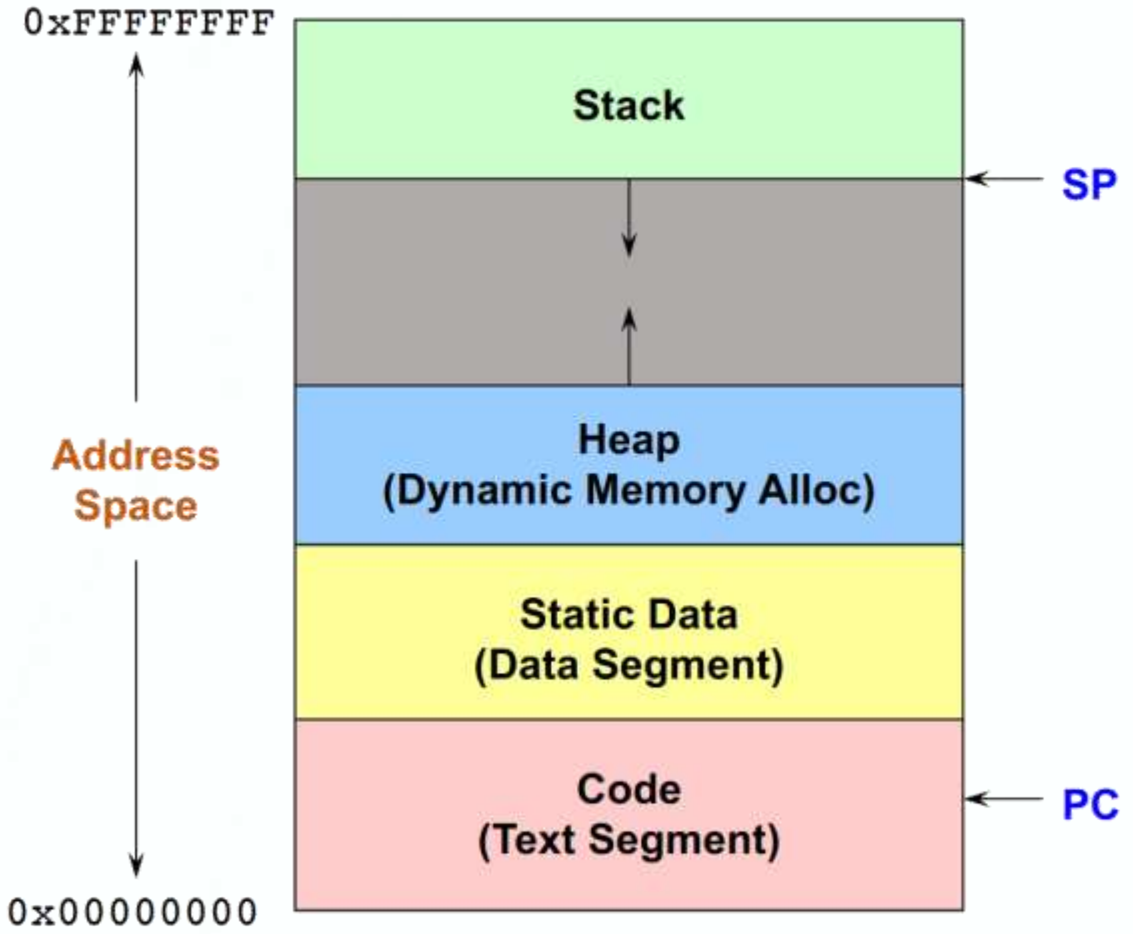
- Really large memory to use
- Linear byte array: [0, N), N roughly 2^{32}, 2^{64}

- *Address Space* ≡ *Protection Domain*
  - OS isolates *Address Spaces*
  - One *Process* can't access another's *Address Space*
  - Same pointer address values in different *Processes* point to different *Physical Memory* locations



Process A *Process* B

Process Address Space



Stack Pointer

```
// works with GCC
register void *sp asm ("sp");
printf("%p", sp);
```

Process **Naming**

A *Process* is named using its *Process ID* (*PID*)

```
top - 20:42:55 up 14 days, 15:45, 14 users,  load average: 1.27, 0.35, 0.12
Tasks: 145 total,   2 running, 141 sleeping,   2 stopped,   0 zombie
%Cpu(s):  5.6 us, 55.3 sy,  0.0 ni,  0.0 id, 35.6 wa,  0.0 hi,  0.0 si,  3.5 st
KiB Mem :  1016140 total,    72288 free,   858408 used,    85444 buff/cache
KiB Swap:        0 total,        0 free,        0 used.    12892 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
   28 root      20   0       0      0      0 R 31.1  0.0  17:11.66 kswapd0
 5546 root      20   0  114980  24596   4732 D 26.2  2.4   0:14.53 check-new-relea
 5583 ryan      20   0   54544   8820   1752 S  2.3  0.9   0:01.24 mosh-server
14294 tomcat    20   0 2497172 230824      0 S  2.0 22.7  16:04.46 java
 5660 ryan      20   0   40536   2172   1488 R  0.7  0.2   0:00.34 top
 8370 mysql     20   0 1123396 199620      0 S  0.3 19.6   8:11.12 mysqld
14074 root      20   0       0      0      0 S  0.3  0.0   0:17.19 kworker/0:2
25217 root      20   0  314032  15260   9204 S  0.3  1.5   2:07.55 php-fpm7.0
25274 parsoid   20   0  937076  28144      0 S  0.3  2.8   5:30.90 nodejs
25292 parsoid   20   0 1049820  50772      0 S  0.3  5.0   6:55.52 nodejs
25313 ghost     20   0 1255612  71152      0 S  0.3  7.0  15:16.30 nodejs
    1 root      20   0  119628   1796      0 S  0.0  0.2   2:39.46 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.17 kthreadd
    3 root      20   0       0      0      0 S  0.0  0.0   0:07.34 ksoftirqd/0
    5 root       0 -20       0      0      0 S  0.0  0.0   0:00.00 kworker/0:0H
    7 root      20   0       0      0      0 S  0.0  0.0   1:01.94 rcu_sched
    8 root      20   0       0      0      0 S  0.0  0.0   0:00.00 rcu_bh
    9 root      rt   0       0      0      0 S  0.0  0.0   0:00.00 migration/0
```

Linux **top** (Table of *Processes*)

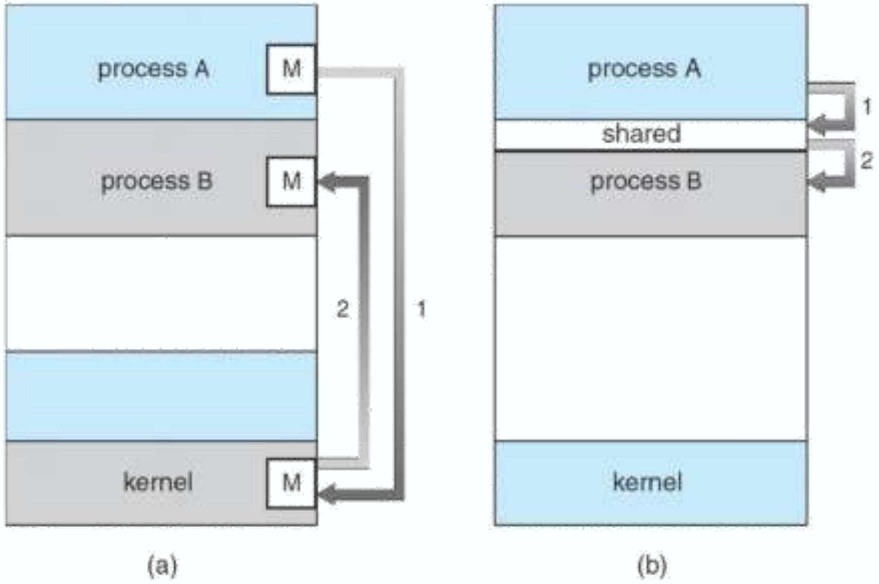
Inter-Process Communication (IPC)

Interaction between *Processes* is sometimes desired

- Conceptually simplest form is through files: **`vim`** edits file, **`gcc`** compiles it
- More complicated: shell /command, Window-manager /App

Real-time interaction is usually required. Methods:

- a) *Message-Passing*
  (through the Kernel)
- b) *Shared Memory*
  (sharing a region of *Physical Memory*)
- c) Asynchronous *Signals* or *Alerts*
  (again through the Kernel)



Kernel & *Processes*

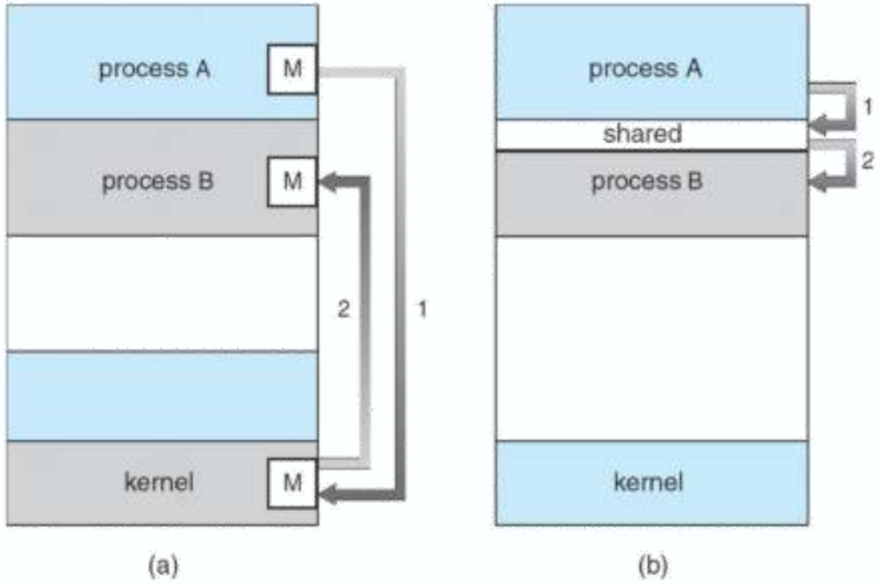
Inter-Process Communication (IPC)

Message-Passing (through the Kernel)

- Good:
  All sharing is explicit → Fewer chances for errors
- Bad:
  Overhead: Data copying. Crossing Protection Domains.

Shared Memory (sharing a region of *Physical Memory*)

- Good:
  Performance. Set up *Shared Memory* once, then access w/o crossing Protection Domains
- Bad:
  Synchronization is required. Things can change behind your back. Error prone.



Inter-Process Communication (IPC)

Unix *PIPE*

Creates a one-way communication channel

`int pipe(int fds[2])`

- **`fds[2]`** is used to return two *File Descriptors*
  - Bytes written to **`fds[1]`** will be read from **`fds[0]`**

How can one *Process* (e.g. reader) know of another *Process*' (e.g. writer) setting-up of a *Pipe* connection?

- **`fork()`** *Process* creation mechanism: (more on this later)

```
int pipefd[2];
pipe(pipefd);
switch( pid = fork() ) {
  case -1: perror("fork error"); exit(1);
  case  0: close(pipefd[0]);  // child process
           // write to pipefd[1]
           break;
  default: close(pipefd[1]);  // parent process
           // read from pipefd[0]
  break;
}
```

Inter-Process Communication (IPC)

Unix *Shared Memory*

```
int shmget(key_t key, size_t size, int shmflg);
```

- Create a *Shared Memory* Segment
- **`key`** : Either a unique identifier returned by **`ftok`** if the Segment can be accessed by other *Processes*, or **`IPC_PRIVATE`** to indicate that the Segment cannot be accessed by other *Processes*
- *Returns* : Identifier of *Shared Memory* Segment associated with the value of the argument **`key`**

```
int shmat(int shmid, const void *addr, int flg);
```

- Attach *Shared Memory* Segment to *Address Space* of the calling *Process*
- **`shmid`** : Identifier returned by **`shmget()`**

```
int shmdt(const void *shmaddr);
```

- Detach calling *Process' Address Space* from *Shared Memory* Segment

Remember: Shared Memory access requires *Synchronization* (more on *Synchronization* Mechanisms later)

Inter-Process Communication (IPC)

UNIX *Signals* (again through the Kernel)

- A very small payload – an integer *Message*
- A fixed set of available OS-defined *Signals*
  - e.g. 9: **`SIGKILL`** , 11: **`SIGSEGV`** , etc.

- Registering an OS signal handler in a *Process*

```
sighandler_t signal(int signum, sighandler_t handler);
```

Note: `typedef void (*sighandler_t)(int);`
i.e. Pointer to void function with a single `int` argument.

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

- Send a *Signal* to a *Process* / *Process Group*

```
int kill(pid_t pid, int sig);
```

Note: If *pid* is positive, then signal *sig* is sent to the *Process* with the ID specified by *pid*.
If *pid* equals 0, then *sig* is sent to every *Process* in the *Process Group* of the calling *Process*.
If *pid* equals -1, then *sig* is sent to every *Process* for which the calling *Process* has permission to send signals, except for *Process* 1 (*init*), but see below.
If *pid* is less than -1, then *sig* is sent to every *Process* in the *Process Group* whose ID is *-pid*.

```
int killpg(int pgrp, int sig);
```

Kernel & *Processes*

Process **Implementation**

A data structure for each process: *Process Control Block* (PCB)

- Contains all information about a *Process*
- PCB is also maintained when the process is not *Running* (why?)

- *Process State*
  - *Running*, *Ready* (*Runnable*), *Waiting*, etc.
- CPU *Registers*
  - `%eip`, `%eax`, etc.
- *Scheduling* information
- *Virtual Memory* mappings
- *I/O* status information
- *Credentials* (*User/Group ID*), *Signal* mask, *Priority*, *Accounting*, etc.

Process state
Process ID
User id, etc.
Program counter
Registers
Address space (VM data structs)
Open files

PCB

The *Process* is a heavyweight *Abstraction*!

struct proc (Solaris OS)

```c
/*
* One structure allocated per active process. It contains all
* data needed about the process while the process may be swapped
* out. Other per-process data (user.h) is also inside the proc structure.
* Lightweight-process data (lwp.h) and the kernel stack may be swapped out.
*/
typedef struct proc {
/*
* Fields requiring no explicit locking
*/
struct vnode *p_exec; /* pointer to a.out vnode */
struct as *p_as; /* process address space pointer */
struct plock *p_lockp; /* ptr to proc struct's mutex lock */
kmutex_t p_crlock; /* lock for p_cred */
struct cred *p_cred; /* process credentials */
/*
* Fields protected by pidlock
*/
int p_swapcnt; /* number of swapped out lwps */
char p_stat; /* status of process */
char p_wcode; /* current wait code */
ushort_t p_pidflag; /* flags protected only by pidlock */
int p_wdata; /* current wait return value */
pid_t p_ppid; /* process id of parent */
struct proc *p_link; /* forward link */
struct proc *p_parent; /* ptr to parent process */
struct proc *p_child; /* ptr to first child process */
struct proc *p_sibling; /* ptr to next sibling proc on chain */
struct proc *p_psibling; /* ptr to prev sibling proc on chain */
struct proc *p_sibling_ns; /* prt to siblings with new state */
struct proc *p_child_ns; /* prt to children with new state */
struct proc *p_next; /* active chain link next */
struct proc *p_prev; /* active chain link prev */
struct proc *p_nextofkin; /* gets accounting info at exit */
struct proc *p_orphan;
struct proc *p_nextorph;
*p_pglink; /* process group hash chain link next */
struct proc *p_ppglink; /* process group hash chain link prev */
struct sess *p_sessp; /* session information */
struct pid *p_pidp; /* process ID info */
struct pid *p_pgidp; /* process group ID info */
/*
* Fields protected by p_lock
*/
kcondvar_t p_cv; /* proc struct's condition variable */
kcondvar_t p_flag_cv;
kcondvar_t p_lwpexit; /* waiting for some lwp to exit */
kcondvar_t p_holdlwps; /* process is waiting for its lwps */
/* to to be held. */
ushort_t p_pad1; /* unused */
uint_t p_flag; /* protected while set. */
/* flags defined below */
clock_t p_utime; /* user time, this process */
clock_t p_stime; /* system time, this process */
clock_t p_cutime; /* sum of children's user time */
clock_t p_cstime; /* sum of children's system time */
caddr_t *p_segacct; /* segment accounting info */
caddr_t p_brkbase; /* base address of heap */
size_t p_brksize; /* heap size in bytes */
/*
* Per process signal stuff.
*/
k_sigset_t p_sig; /* signals pending to this process */
k_sigset_t p_ignore; /* ignore when generated */
k_sigset_t p_siginfo; /* gets signal info with signal */
struct sigqueue *p_sigqueue; /* queued siginfo structures */
struct sigqhdr *p_sigqhdr; /* hdr to sigqueue structure pool */
struct sigqhdr *p_signhdr; /* hdr to signotify structure pool */
uchar_t p_stopsig; /* jobcontrol stop signal */
```

`struct proc` (Solaris OS)

```c
/*
* Special per-process flag when set will fix misaligned memory
* references.
*/
char p_fixalignment;
/*
* Per process lwp and kernel thread stuff
*/
id_t p_lwpid; /* most recently allocated lwpid */
int p_lwpcnt; /* number of lwps in this process */
int p_lwprcnt; /* number of not stopped lwps */
int p_lwpwait; /* number of lwps in lwp_wait() */
int p_zombcnt; /* number of zombie lwps */
int p_zomb_max; /* number of entries in p_zomb_tid */
id_t *p_zomb_tid; /* array of zombie lwpids */
kthread_t *p_tlist; /* circular list of threads */
/*
* /proc (process filesystem) debugger interface stuff.
*/
k_sigset_t p_sigmask; /* mask of traced signals (/proc) */
k_fltset_t p_fltmask; /* mask of traced faults (/proc) */
struct vnode *p_trace; /* pointer to primary /proc vnode */
struct vnode *p_plist; /* list of /proc vnodes for process */
kthread_t *p_agenttp; /* thread ptr for /proc agent lwp */
struct watched_area *p_warea; /* list of watched areas */
ulong_t p_nwarea; /* number of watched areas */
struct watched_page *p_wpage; /* remembered watched pages (vfork) */
int p_nwpage; /* number of watched pages (vfork) */
int p_mapcnt; /* number of active pr_mappage()s */
struct proc *p_rlink; /* linked list for server */
kcondvar_t p_srwchan_cv;
size_t p_stksize; /* process stack size in bytes */
```

```c
/*
* Microstate accounting, resource usage, and real-time profiling
*/
hrtime_t p_mstart; /* hi-res process start time */
hrtime_t p_mterm; /* hi-res process termination time */
hrtime_t p_mlreal; /* elapsed time sum over defunct lwps */
hrtime_t p_acct[NMSTATES]; /* microstate sum over defunct lwps */
struct lrusage p_ru; /* lrusage sum over defunct lwps */
struct itimerval p_rprof_timer; /* ITIMER_REALPROF interval timer */
uintptr_t p_rprof_cyclic; /* ITIMER_REALPROF cyclic */
uint_t p_defunct; /* number of defunct lwps */
/*
* profiling. A lock is used in the event of multiple lwp's
* using the same profiling base/size.
*/
kmutex_t p_pflock; /* protects user profile arguments */
struct prof p_prof; /* profile arguments */
/*
* The user structure
*/
struct user p_user; /* (see sys/user.h) */
/*
* Doors.
*/
kthread_t *p_server_threads;
struct door_node *p_door_list; /* active doors */
struct door_node *p_unref_list;
kcondvar_t p_server_cv;
char p_unref_thread; /* unref thread created */
```

struct proc (Solaris OS)

```
/*
* Kernel probes
*/
uchar_t p_tnf_flags;
/*
* C2 Security (C2_AUDIT)
*/
caddr_t p_audit_data; /* per process audit structure */
kthread_t *p_aslwptp; /* thread ptr representing "aslwp" */
#if defined(i386) || defined(__i386) || defined(__ia64)
/*
* LDT support.
*/
kmutex_t p_ldtlock; /* protects the following fields */
struct seg_desc *p_ldt; /* Pointer to private LDT */
struct seg_desc p_ldt_desc; /* segment descriptor for private LDT */
int p_ldtlimit; /* highest selector used */
#endif
size_t p_swrss; /* resident set size before last swap */
struct aio *p_aio; /* pointer to async I/O struct */
struct itimer **p_itimer; /* interval timers */
k_sigset_t p_notifsigs; /* signals in notification set */
kcondvar_t p_notifcv; /* notif cv to synchronize with aslwp */
timeout_id_t p_alarmid; /* alarm's timeout id */
uint_t p_sc_unblocked; /* number of unblocked threads */
struct vnode *p_sc_door; /* scheduler activations door */
caddr_t p_usrstack; /* top of the process stack */
uint_t p_stkprot; /* stack memory protection */
model_t p_model; /* data model determined at exec time */
struct lwpchan_data *p_lcp; /* lwpchan cache */
/*
* protects unmapping and initilization of robust locks.
*/
kmutex_t p_lcp_mutexinitlock;
utrap_handler_t *p_utraps; /* pointer to user trap handlers */
refstr_t *p_corefile; /* pattern for core file */
#if defined(__ia64)
caddr_t p_upstack; /* base of the upward-growing stack */
size_t p_upstksize; /* size of that stack, in bytes */
uchar_t p_isa; /* which instruction set is utilized */
#endif
void *p_rce; /* resource control extension data */
struct task *p_task; /* our containing task */
struct proc *p_taskprev; /* ptr to previous process in task */
struct proc *p_tasknext; /* ptr to next process in task */
int p_lwpdaemon; /* number of TP_DAEMON lwps */
int p_lwpdwait; /* number of daemons in lwp_wait() */
kthread_t **p_tidhash; /* tid (lwpid) lookup hash table */
struct sc_data *p_schedctl; /* available schedctl structures */
} proc_t;
```

Kernel & *Processes*

Process State

A process has an *Execution State* to indicate what it is doing

Running

Executing *Instructions* on the CPU

- It is the *Process* that has control of the CPU

Ready

Waiting to be assigned to the CPU

- Ready to execute, but another *Process* is executing on the CPU

Waiting

Waiting for an Event, e.g., I/O completion

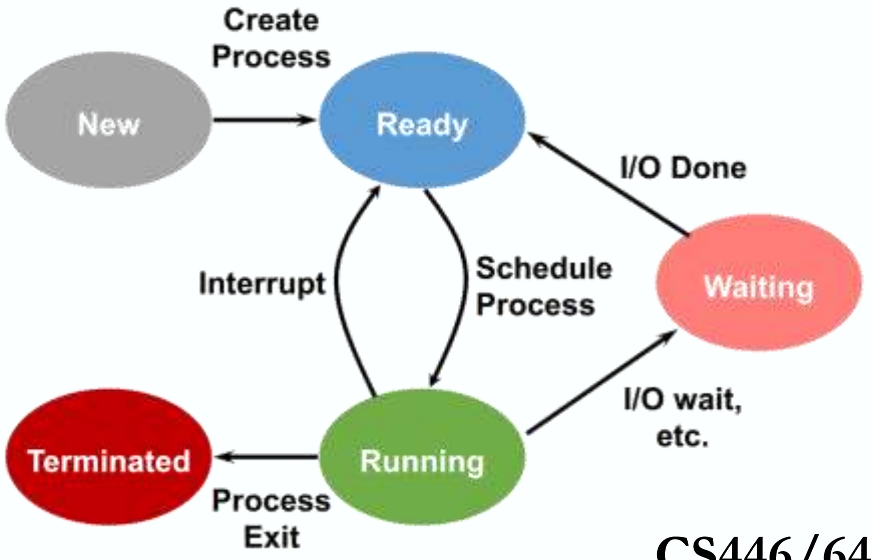
- It cannot make progress until event is signaled (e.g. disk completes)

Transition of *Process State*

As a *Process* executes, it moves from *Execution State* to *Execution State*

- In Unix **`ps`**: **`STAT`** column indicates *Execution State*
  - Maximum number of *Processes* in OS: "In the Linux *Kernel Space* context, a *Process* and a *Thread* are one and the same. They're handled the same way by the Kernel. They both occupy a slot in the **`task_struct`**. A *Thread*, by common terminology, is in Linux a *Process* that shares *Resources* with another *Process* (they will also share a *Thread Group ID*)."
    - **`/proc/sys/kernel/threads-max`**
    - **`/proc/sys/kernel/pid_max`**

General
Process State Graph:

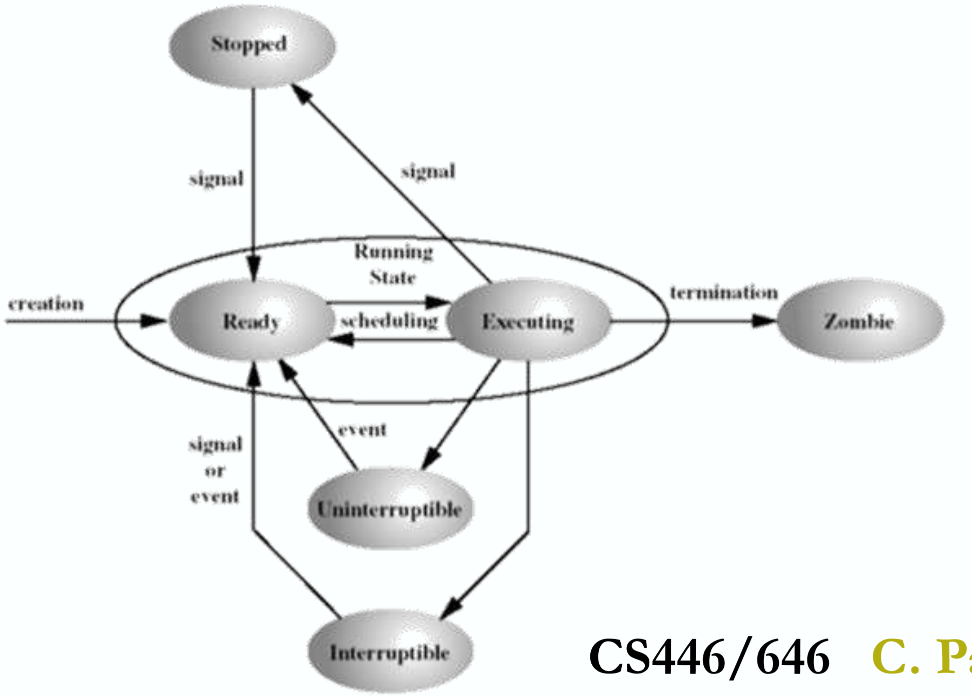


Transition of *Process State*

As a *Process* executes, it moves from *Execution State* to *Execution State*

- In Unix **`ps`**: **`STAT`** column indicates *Execution State*
  - Maximum number of *Processes* in OS: "In the Linux *Kernel Space* context, a *Process* and a *Thread* are one and the same. They're handled the same way by the Kernel. They both occupy a slot in the **`task_struct`**. A *Thread*, by common terminology, is in Linux a *Process* that shares *Resources* with another *Process* (they will also share a *Thread Group ID*)."
    - **`/proc/sys/kernel/threads-max`**
    - **`/proc/sys/kernel/pid_max`**

Linux
Process State Graph:



State Queues

How the OS keeps track of *Processes*

Simple 1st Idea:

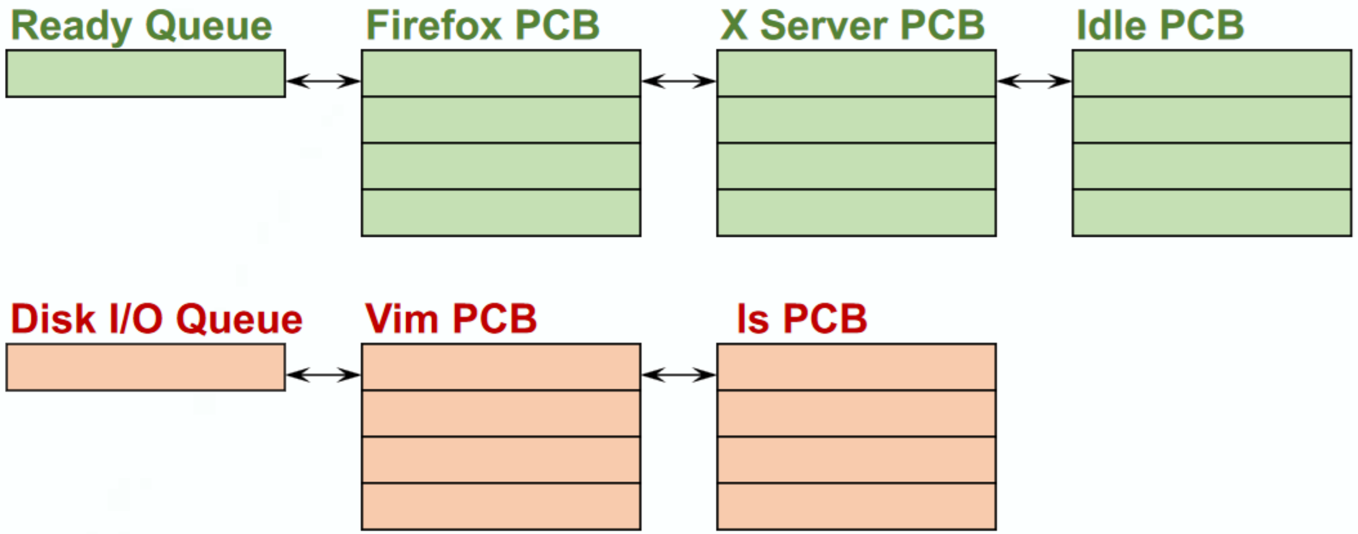
- List of *Processes*
- How to find out ones in the *Ready* state?
  - Iterating through the list is slow

Improvement:

- Partition List of *Processes* based on type of state
- OS maintains a collection of *Queues* that represent the state of all *Processes*
  - A typical implementation is to have one *Queue* for each type of state: *Ready*, *Waiting*, etc.
- Each *Process Control Block* (*PCB*) is queued on a *State Queue* according to its current *State*
  - When a *Process* changes state, its *PCB* is moved from one *State Queue* into another

Kernel & *Processes*

State Queues



Console Queue

Sleep Queue

·

·

·

Note: There may be many *Wait* state *Queues*, one for each type of *Wait* (disk, console, timer, network, etc.)

Scheduling

How the OS determines which *Process* to run

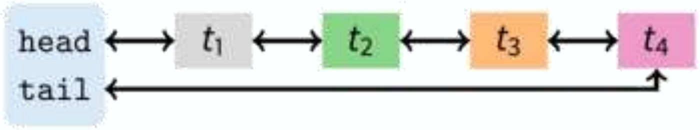
- If 0 *Runnable*, run idle loop (or Halt CPU), if 1 *Runnable*, run that one
- If >1 *Runnable*, must make *Scheduling* decision

1st Idea: Scan *Process Table* for first *Runnable* one

- Expensive & Unfair (smaller *PID*s prioritized)

Better Idea: *First-In First-Out* (FIFO) *Scheduling*

- Put tasks on back of *Queue*, pull them from front:
  - Pintos does this – see **`ready_list in thread.c`**



Even Better: *Priority Scheduling*

- More on that in a dedicated upcoming Lecture…

Process **Dispatching Mechanism**

OS *Process* Dispatching loop

I. Gaining Control

```
while(1) {
  // run process for a while;
  // save process state;
  // next process = schedule (ready processes);
  // load next process state;
}
```

II. *Context Switching*

Kernel & *Processes*

I. How to Gain Control

- First of all, must switch from *User Mode* to *Kernel Mode*

Cooperative Multitasking:
Processes voluntarily *yield* control back to OS
- How / When? With *System Calls* that relinquish CPU
  - Trustworthiness: OS needs to trust User *Processes*

True Multitasking:
OS "*Preempts*" *Processes* by periodic *Interrupts*
- Processes are assigned *Time Slices* (/"*Quanta*") of execution
- Dispatcher counts Timer *Interrupts* before context switch
  - OS needs to trust no one

I. How to Gain Control

Preemption

- Temporarily interrupting an executing *Task*, with the intention of resuming it later.
  Interruption performed by an external *Preemptive Scheduler* with no assistance/cooperation from the *Task*.

Process Scheduling decision triggering

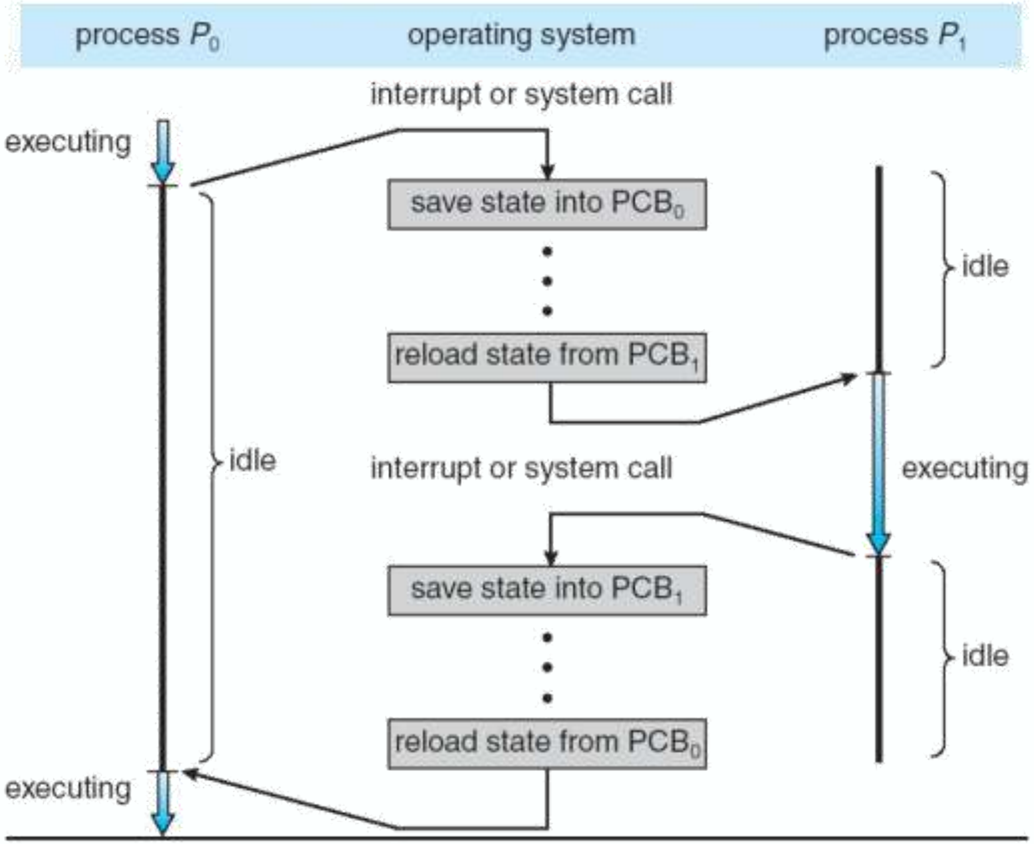
- ***Cooperative Multitasking:*** *Yielding* control of CPU
  - Voluntarily, e.g. `sched_yield`
    Note: Cause the calling *Thread* to relinquish the CPU. The *Thread* is moved to the end of the *Queue* for its Static Priority and a new *Thread* gets to run.
  - Forced, on a *System Call*, *Page Fault*, *Illegal Instruction*, etc.
- ***True Multitasking:*** *Preemption*
  - Periodic *Timer Interrupt*
    - Indicates that *Running Process* used up *Time Slice* (/"*Quantum*"), schedule another
      - *Time Slice:* The period of time for which a *Process* is allowed to run in a *Preemptive Multitasking* system
  - (Device) *Interrupt*
    - Disk request completed / *Packet* arrived on Network
      - *Process* previously *Waiting* now becomes *Runnable*

Process Dispatching Mechanism

II. How to Switch *Context*

Context Switching

- The procedure of storing the *State* of a *Process* or *Thread*, so that it can be restored and resume execution at a later point, and then restoring a different, previously saved, *State*.



Changing over the running *Process*

- CPU (& MMU) Hardware state is changed from one to another
  - Can happen thousands of times each second

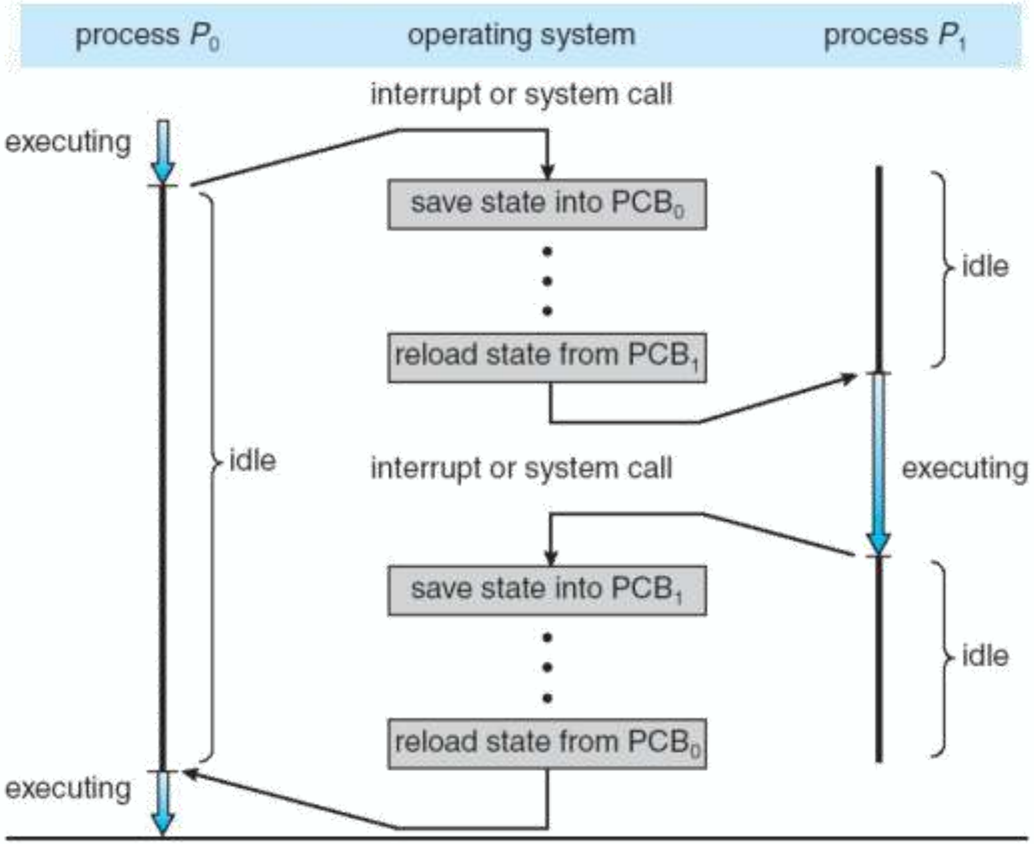
Process Dispatching Mechanism

II. How to Switch *Context*

Context Switching

Changing over the running *Process*

- Save P0's *User Mode* CPU *Context* and switch from *User* to *Kernel Mode* (HW)
- Handle *System Call* or *Interrupt* (OS)
- Save P0's *Kernel Mode* CPU *Context* and switch to *Scheduler* CPU *Context* (OS + HW)
- *Scheduler* selects another *Process* P1 (OS)
- Switch to P1's *Address Space* (OS + HW)
- Save *Scheduler* CPU *Context* and switch to P1's *Kernel Mode* CPU *Context* (OS + HW)
- Switch from *Kernel Mode* to *User Mode* and load P1's *User Mode* CPU *Context* (HW)



***Process* Dispatching Mechanism**

II. How to Switch *Context*

Context Switching is very machine-dependent. Typical things include:

- Save *Program Counter* (PC) and *Integer Registers* (always)
- Save *Floating Point Registers* or other *Special Registers*
- Save *Condition Codes* (e.g. in **`EFLAGS`** *Register*)
- Change *Virtual Address Translations*

Tricky:

- Need to save *Program Counter* (PC) to the *Process Control Block* (PCB) in *Memory*
  - But that would require to load the PC of code that saves the original PC, which would now be **lost**
- Need to save all *Registers* to the *Process Control Block* (PCB) in *Memory*
  - But we have to actually run code to save *Registers*, and running code **changes** *Register* values
- Need combined Software/Hardware support
  - CPU will save the *Program Counter* (PC) when an *Interrupt* occurs, at a known location (typically onto the *CPU Stack*)

Process Dispatching Mechanism

II. How to Switch *Context*

Context Switching has a non-negligible cost

- It represents pure overhead: The system does no useful work while *Context Switching*
  - Save/restore *Floating Point Registers* expensive
    - Optimization: only save if *Process* actually used floating point arithmetic
  - May require flushing / *Invalidating* part of the MMU's *Translation Lookaside Buffer* (*TLB*) cache
    - Memory cache that stores the recently utilized translations of *Virtual Memory* to *Physical Memory*. Used to reduce the time taken to access a user *Memory* location.
- The OS must balance *Context Switching* frequency with *Scheduling* requirements

Context Switching usually causes more CPU *Cache Misses*

- Due to frequent switching between *Working Sets*
  - *Working Set:* The *(Memory) Pages* most frequently accessed at some point
  - Potentially causing *Cache Pollution*
- Also, in the extreme case of overcommitted resources: [*Thrashing*]

Process-related *System Calls*

Allow one *Process* to create another *Child Process*

A *Process* is created by another *Process*

- *Parent* is creator, *Child* is created (Unix: `ps` "*PPID*" field / `pstree -p` command)
- Very first *Process* (Unix: `init` / Linux: [`systemd`]) (*PID* 1 and *PPID* 0))

  "Once the Kernel has started, it starts the `init` process, a daemon which then bootstraps the *User Space*, for example by checking and mounting file systems, and starting up other *Processes*. The `init` system is the first *Daemon* to start (during booting) and the last *Daemon* to terminate (during shutdown)."

  Note: `init` is a *User Space Process*

Parent defines Resources and Privileges for its *Children*

- Unix: *Process User ID* is inherited – *Children (Processes)* of your *Shell* (*Process*) will execute with the same *User Privileges*

After creating a *Child Process*

- *Parent Process* may either wait for it to finish its task, or concurrently continue its work

Process Creation in Windows

```
CreateProcess( prog,      // Module to load (e.g. "c:\\winnt\\notepad.exe" or NULL – use command line)
               argv[1],   // Command line args
               NULL,      // Process handle not inheritable
               NULL,      // Thread handle not inheritable
               FALSE,     // Set handle inheritance to FALSE
               0,         // No creation flags
               NULL,      // Use parent's environment block
               NULL,      // Use parent's starting directory
               &si,       // Pointer to STARTUPINFO structure
               &pi )      // Pointer to PROCESS_INFORMATION structure
```

1. Creates and initializes a new PCB
2. Creates and initializes a new *Address Space*
3. Loads the program specified by **`prog`** into the *Address Space*
4. Copies **`args`** into Memory allocated in *Address Space*
5. Initializes the saved hardware *Context* to start execution at **`main`** (or as specified)
6. Places the *PCB* on the *Ready Queue*

Note: Returns **`BOOL`**. If **`CreateProcess`** succeeds, it returns a **`PROCESS_INFORMATION`** structure that contains handles and identifiers for the new *Process* and its primary *Thread*. The *Thread* and *Process* Handles are created with full access rights, although you can restrict access if you specify security descriptors.

Process-related *System Calls*

Process Creation in Unix

`pid_t fork (void)`

1. Creates and initializes a new PCB
2. Creates a new *Address Space*
3. Initializes the *Address Space* with a **copy** of the *Address Space* of the *Parent*
4. Initializes the Kernel *Resources* to point to the *Parent*'s Resources (e.g. open *Files*)
5. Places the *PCB* on the *Ready Queue*

`fork()` returns **twice**:

(???)

Because if it's successful, we now have 2 *Processes*

- Returns the *Child*'s *PID* to the *Parent Process*
- Returns 0 to the *Child Process*

Process Creation in Unix

pid_t fork (void)

SYNOPSIS

```
#include <unistd.h>

pid_t
fork(void);
```

DESCRIPTION

Fork() causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an lseek(2) on a descriptor in the child process can affect a subsequent read or write by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.
- The child processes resource utilizations are set to 0; see setrlimit(2).

RETURN VALUES

Upon successful completion, **fork**() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable *errno* is set to indicate the error.

ERRORS

Fork() will fail and no child process will be created if:

[EAGAIN]	The system-imposed limit on the total number of processes under execution would be exceeded. This limit is configuration-dependent.
[EAGAIN]	The system-imposed limit MAXUPRC (<*sys/param.h*>) on the total number of processes under execution by a single user would be exceeded.
[ENOMEM]	There is insufficient swap space for the new process.

Process Creation in Unix

`pid_t fork (void)`

```c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
  char *name = argv[0];
  int child_pid = fork();
  if (child_pid == 0) {
    printf("I'm the child of %s and my pid is: %d\n", name, getpid());
    return 0;
  } else {
    printf("I'm the parent and my child's pid is: %d\n", child_pid);
    return 0;
  }
}
```

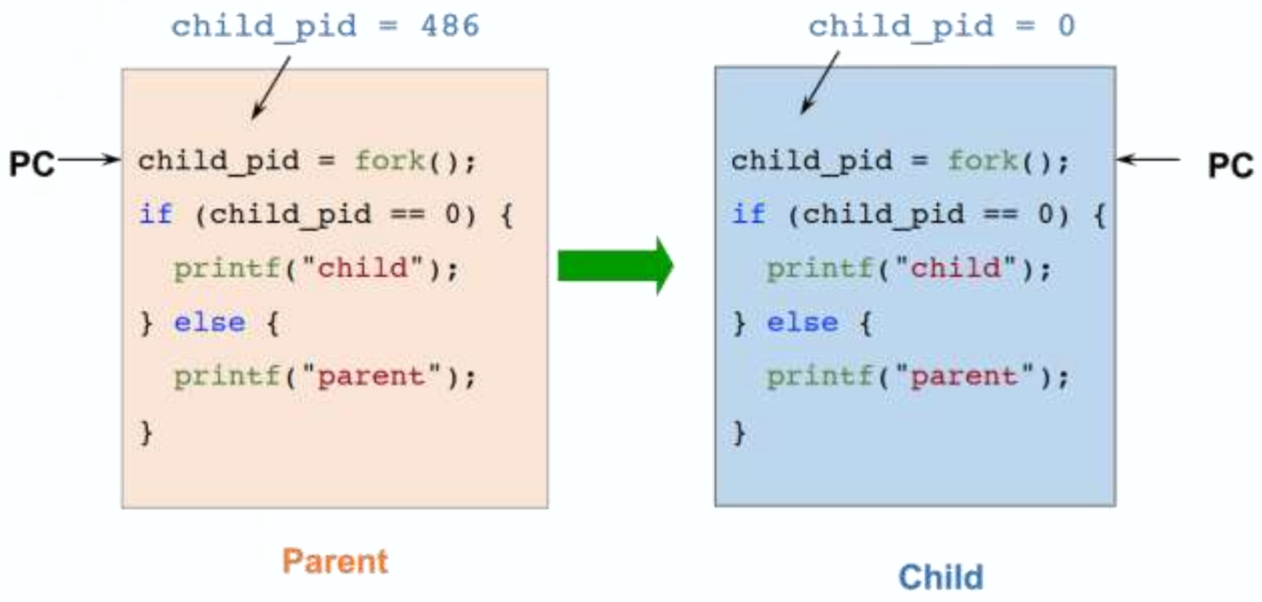
Expected program output (`$ gcc -o fork fork.c && ./fork`)?

```
I'm the parent and my child's pid is: 486
I'm the child of ./fork and my pid is: 486
```

***Process* Creation in Unix**

`pid_t fork (void)`

➢ After duplicating *Address Spaces*

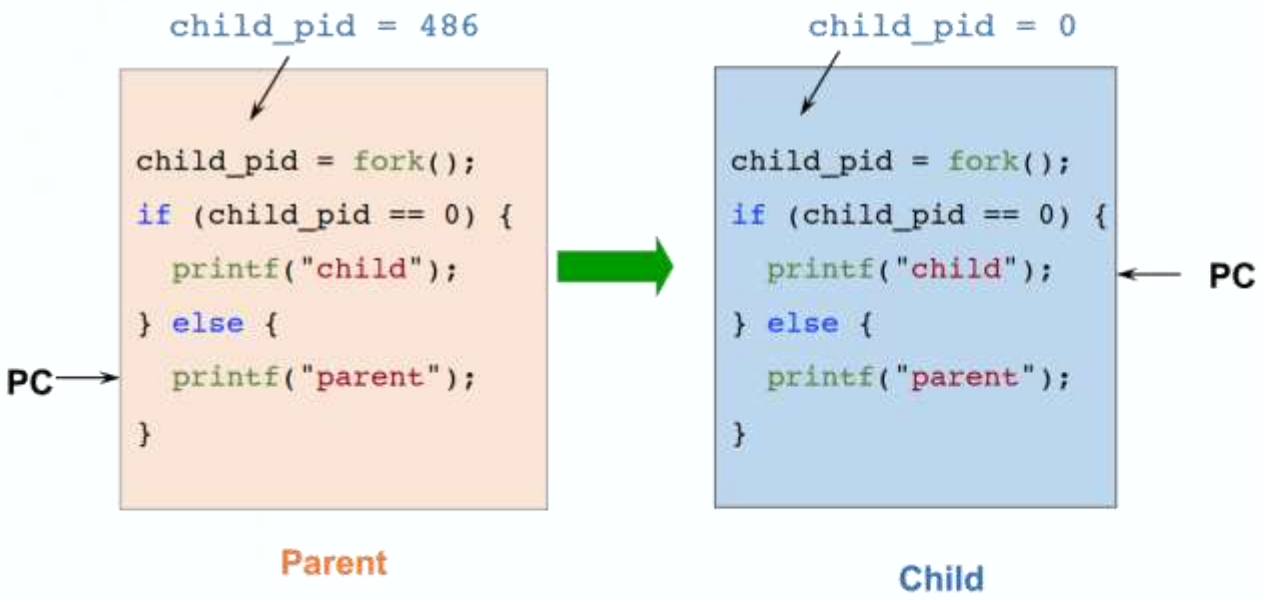


The hardware contexts stored in the PCBs of the two processes will be identical, meaning the `EIP` register will point to the same instruction

***Process* Creation in Unix**

`pid_t fork (void)`

➢ Divergence between different *Processes*



Process Creation in Unix

`pid_t fork (void)`

Program output (continued)

```
$ ./fork
My child is 486
Child of ./fork is 486
$ ./fork
Child of ./fork is 486
My child is 486
```

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
  char *name = argv[0];
  int child_pid = fork();
  if (child_pid == 0) {
    printf("Child of %s is %d\n", name, getpid());
    return 0;
  } else {
    printf("My child is %d\n", child_pid);
    return 0;
  }
}
```

Note: Notice possible change of output order between subsequent runs of the same executable

Process **Creation in Unix**

Running a program

```
int execv(char *path, char *argv[]);
int execve(const char *filename, char *const argv[], char *const envp[]);
int execvp(const char *file, char *const argv[]);
```

`v` : Takes an **`argv`** array of C-strings populated with the Program, its Arguments/Flags, and a **`NULL`**-pointer at the end

`e` : Takes an **`envp`** array of C-strings populated Environment variables. `p` : Inherits the *Parent Process* Environment.

`execXX(...)`

1. Stops the Program executing under the current *Process*
2. Loads the new Program **`prog`** into the calling *Process' Address Space*
3. Initializes Hardware *Context* and *Args* for the new Program
4. Places the PCB onto the *Ready Queue*

Note: It does *NOT* create a new *Process*. "… it **replaces** the entire current *Process* with a new Program.

It loads the Program into the current *Process' Address Space* and runs it from the entry point.

PintOS **`exec`** is more like a combined **`fork`**/**`exec`**

➢ What does it mean for **`exec`** to return?

Process-related *System Calls*

Process Creation in Unix

Most calls to **`fork()`** will be followed by **`execXX(…)`**

- Can always be combined in a single **`spawn`** *System Call*

Very useful when

- *Child* is working together with *Parent*
- *Child* relies on *Parent's* data to accomplish its task

Example: Web Server

Note: The **`accept()`** call creates a new *Socket Descriptor* with the same properties as **`server_sock`** and returns it to the caller. If the queue has no pending connection requests, **`accept()`** *blocks* the caller unless **`server_sock`** is in nonblocking mode. If no connection requests are queued and **`server_sock`** is in nonblocking mode, **`accept()`** returns -1 and sets the error code to **`EWOULDBLOCK`**.

```
while (1) {
  int client_sock = accept(server_sock, addr, addrlen);
  if ((fork_pid = fork()) == 0) {  // Child Process
    // Handle client request
  } else {  // Parent Process
    // Close server socket
  }
}
```

Process Creation in Unix

Example: Simple Shell

```
pid_t pid; char **av;

void execProc () {
  execvp (av[0], av);
  perror (av[0]);
  _exit (-1);
}

/* ... main loop: */
  for (;;) {
    parseInput (&av, stdin); //read shell commands from stdin
    switch (fork_pid = fork ()) {
      case -1:  //Parent Process – fork error
        perror ("fork error"); break;
      case 0:  //Child Process
        execProc (); break;
      default:  //Parent Process
        waitpid (pid, NULL, 0); break;
    }
  }
```

```
$ gcc -o simpleshell simpleshell.c
$ ./simpleshell
$ date
Wed Sep  6 09:20:53 PDT 2023
$ /usr/bin/gcc --version
gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying
conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE.
…
```

Process-related *System Calls*

Process Creation in Unix

Majority of calls to **`fork()`** are followed by **`execXX(…)`**
- Can always be combined in a single **`spawn`** *System Call*

Very useful when
- *Child* is working together with *Parent*
- *Child* can rely on *Parent's* data to accomplish its task

Real win is in the simplicity of its interface
- Many things we might want to do to the *Child Process*
  - Manipulate *File Descriptors*, set *Environment Variables*, reduce Privileges, etc.
- Yet **`fork()`** requires no arguments at all
  - *Remember:* Windows **`CreateProcess(…)`** *System Call* required a lot of different options (e.g. function call arguments) to cover the possibilities for the new Program we intend to run

Process Creation in Unix

Example: Simple Shell with Redirection

```
void execProc (void) {
  int fd;
  if (infile) { /* non-NULL for "command < infile" */
    if ((fd = open (infile, O_RDONLY)) < 0) {
      perror (infile);
      exit (1);
    }
    if (fd != 0) {
      dup2 (fd, 0);
      close (fd);
    }
  }
  /*...do same for outfile→fd:1, errfile→fd:2...*/
  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}
```

Note:
0: STDIN_FILENO
1: STDOUT_FILENO
2: STDERR_FILENO

```
$ gcc -o redirshell redirshell.c
$ ./redirshell
$ ls > list.txt
$ sort < list.txt > sorted_list.txt
$ cat sorted_list.txt
a.c
b.c
cs446.txt
…
```

Process Creation in Unix

Why Windows uses **`CreateProcess`** while Unix uses **`fork`**/**`execXX`**?

- Different OS design philosophy

- What happens if you run **`exec csh`** in a *Shell*?

- What happens if you run **`exec ls`** in a *Shell*?

- **`fork()`** may return an error. Why might this happen?
  - There exists an **`explain_fork()`** to obtain explanations for errors returned by the **`fork()`** *System Call*

Process-related *System Calls*

Process **(Normal) Termination**

In Unix: **`exit (int status)`** (In Windows: **`ExitProcess (int status)`**)

Frees Resources and terminates, returns **`status & 0377`**$_8$ to *Parent* **`wait()`**

1. Terminate all *Threads* (future Lecture)
2. Close open *Files*, Network Connections
3. Allocated *Memory* (mark *Virtual Memory Pages* as Free)
4. Remove PCB from Kernel data structures, delete

Note: All functions registered with **`atexit()`** and **`on_exit()`** are called, in the reverse order of their registration. … All open **`stdio`** streams are flushed and closed. Files created by **`tmpfile()`** are removed.

A *Process* does not clean up after itself

- The OS has to do it (Why?)
  - *Virtual Memory* Mappings, *Resource* allocation, etc. falls under *Kernel Space* responsibilities

Process **(Immediate) Termination**

In Unix: **`_exit (int status)`**

Frees resources and terminates "***immediately***", returns **`status & 0377`**$_8$ to *Parent* **`wait()`**

Note: Terminates the calling *Process* "immediately". Any open *File Descriptors* belonging to the *Process* are closed; any *Children* of the *Process* are inherited by *Process* 1, **`init`**, and the *Process's Parent* is sent a **`SIGCHLD`** *Signal*.

I.e. calling **`exit()`** from a **`fork`**ed *Child* that hasn't successfully **`exec`**'d something (to replace the original *Process* image it inherits), will interfere with the *Parent Process*' external data (files) by calling its **`atexit`** handlers, calling its *Signal* handlers, and/or flushing buffers (i.e. print out whatever exists in **`stdio`** of the newly **`fork`**ed *Child Process* which is an exact *User Space* duplicate of the *Parent* – "*double-flushing*")

Rule-of-Thumb:
- Use **`_exit()`** to abort the *Child* when the **`exec`** fails
- Use **`_exit()`** to terminate a *Child* that performs no **`exec`** (more rare)

Waiting on a ***Process***

In Unix: **`pid_t wait (int *status)`** (In Windows: **`WaitForSingleObject(eHandle,millis)`**)

- Suspends the current *Process* until **any** *Child Process* changes State (*Remember:* Linux *Process* States).

`pid_t waitpid (pid_t pid, int *status, int options)`

- Suspends the current *Process* until the **`pid` - specified** *Child Process* changes State.

Note:
`pid` > 0: wait for the *Child* whose *Process ID* is equal to the value of **`pid`**.
`pid` = 0: wait for any *Child Process* whose *Process Group ID* is equal to that of the calling *rocess*.
`pid` = -1: wait for any *Child Proceds*.
`pid` < 0: wait for any *Child Process* whose *Process Group ID* is equal to the absolute value of **`pid`**.

Return value of **`wait`**/**`waitpid`**:

- **`wait()`**:
  On success, returns the PID of the State-changed (e.g. terminated) *Child*; on error, -1 is returned.
- **`waitpid()`**:
  On success, returns the PID of the *Child* whose State has changed; if **`WNOHANG`** was specified and one or more *Child*(*ren*) specified by PID exist, but have not yet changed State, then 0 is returned.
  On error, -1 is returned.

Waiting on a *Process*

The purpose of the **`wait()`** *syscall* is to allow the *Child Process* to report a status back to the *Parent Process*

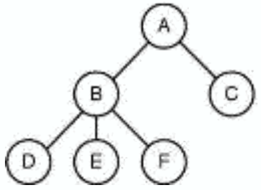
- *Child Process* is not completely cleaned up when it **`exit()`**s
  It "dies" as a *Running Process*, most *Resources* are released, but it still remains in the *Process Table*
  - That's where its *Exit Status* is stored, so that the *Parent* can retrieve it by calling one of the **`wait()`** variants
- It will remain & keep consuming that *Process Table* slot until "*Reaped*" by being **`wait()`**ed on
- Unix: Every *Process* **must** be "*Reaped*" by a *Parent*

Note: Usual **convention** is that the *Parent* that **`fork`**ed the *Children* **`wait`**s on them, because typically they are doing part of the job it was expected to do.

Zombie Process: *Child* that terminates, but has not been **`wait()`**ed for yet (by the *Parent*). The Kernel maintains a minimal set of information about the zombie (PID, termination status, resource usage information) in order to allow the *Parent* to later perform a **`wait()`** to obtain information about the *Child*. As long as a zombie is not removed from the system via a **`wait()`**, it will consume a slot in the kernel *Process Table*, and if this table fills, it will not be possible to create further *Processes*. If a *Parent Process* terminates, then its *Zombie Children* (if any) are adopted by **`init()`**, which automatically performs a **`wait()`** to remove the *Zombies*.

Note: Every *Process* spawned after **`init`** has a *Parent Process*

- Each *Parent* can have many *Children* and those can have their own *Children*
- *Zombie Processes* eventually "adopted" by **`init`**, which will routinely **`wait()`** on and *Reap* any *Zombie Children*



A *Process Tree*

Waiting on a ***Process***

The purpose of the **`wait()`** *syscall* is to allow the *Child Process* to report a status back to the *Parent Process*

- *Child Process* is not completely cleaned up when it **`exit()`**s

  It "dies" as a *Running Process*, most *Resources* are released, but it still remains in the *Process Table*

  - That's where its *Exit Status* is stored, so that the *Parent* can retrieve it by calling one of the **`wait()`** variants
- It will remain & keep consuming that *Process Table* slot until "*Reaped*" by being **`wait()`**ed on
- Unix: Every *Process* **must** be "*Reaped*" by a *Parent*

Note: Usual **convention** is that the *Parent* that **`fork`**ed the *Children* **`wait`**s on them, because typically they are doing part of the job it was expected to do.

Orphan Process:

When a *Parent Process* dies before a *Child Process*.

- In this case, the Kernel knows that it's not going to get a **`wait()`** call.
- The Kernel will immediately put *Orphan Processes* under the care of **`init`**, which will routinely **`wait`** on and *Reap* them eventually.
  - Try it:

```
ping -D localhost >/dev/null 2>&1 &      # Terminal 1: Start looping ping as daemon (background)
pstree -p                                # Terminal 2: Find PID of ping’s shell parent process
kill -SIGTERM [PING_PARENT_SHELL_PID]    # Terminal 2: Kill ping’s shell parent process (T1 dies)
pstree -p                                # see how ping has been reparented to systemd (init)
```

Process Miscellanea

- A *Process* can be **`kill()`**ed
  - Not just by a *Parent Process*
  - Any *Process* running under the same *User ID* or as the **`root`** *User* can **`kill()`** any other *Process*
  - Terminal command sequence Ctrl-C can be used to send **`SIGINT`** *Signal* to the active *Process* (in that terminal)
    - Will (by default – unless otherwise handled) terminate that *Process*
  - Terminal command sequence Ctrl-Z can be used to send **`SIGTSTP`** *Signal* to the active *Process* (in that terminal)
    - Will (by default – unless otherwise handled) pause that *Process* and move it to background
    - Command **`jobs`** can be used to see such stopped (alive, just not running) background *Processes*
    - Typing **`fg`** in shell will resume it in foreground, **`bg`** will resume it in background
- When a *Parent Process* is **`kill()`**ed, it doesn't by default **`kill()`** the *Children Processes*
  - You can do that by **`kill()`**ing a *Process Group*
    - use with (negative) Process Group ID (PGID)
    - Or use **`killpg()`**
  - As mentioned if the *Parent Process* is **`kill()`**ed first, it ill leave *Orphan Processes* out of any *Children Processes*

CS-446/646

Time for Questions !