1/10

Friday, January 12, 2024 9:56 AM

vectored subroutine call uses memory in table for an event

interrupt interrupts fetch-execute cycle by going to another place in memory

fault: some piece of executing code did something incorrect (i.e. divide by 0) - this is "segmentation fault (core dumped)"

traps: a user triggered event, like a subroutine call (except you go to trap table instead)

tables get populated by bootloader

CPU privilege mode: you can run any assembly instructions you want (you go to physical addresses)

traps, interrupts, and faults elevate privilege and transfer control to the subroutine as one "atomic" step that cannot be interrupted

1/12

Friday, January 12, 2024 9:56 AM

1. get the build tools
2. unpack sources
3. get the config file and run through editor

3.5

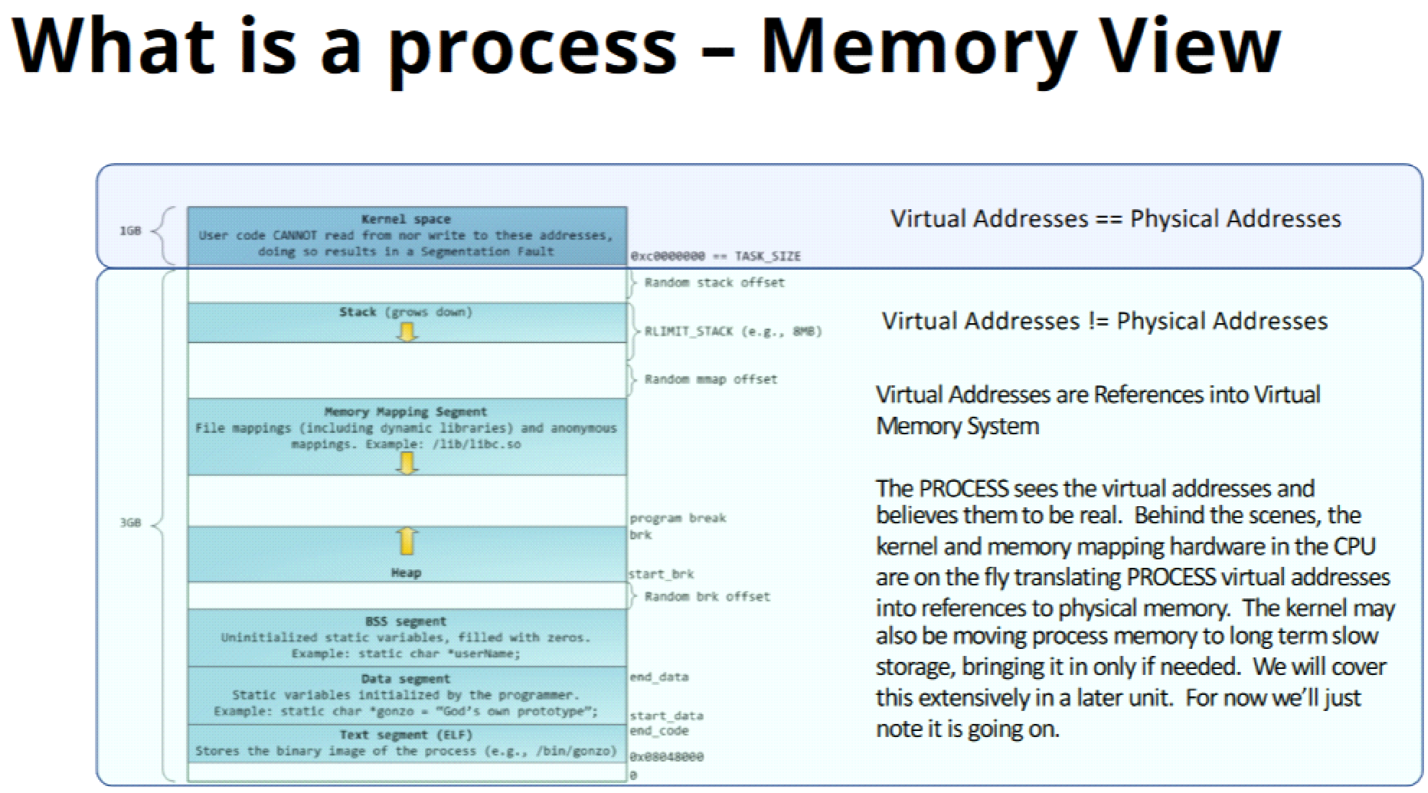
4. "make -j4" and wait forever
    - make (turn source code into built binaries), kernels, make modules install (gives us modules, do sudo to put in sys directory)
    - binaries go to /boot
    - modules are like plugins
    - if uninstalling kernel: leave build directory alone (that way we don't have to rebuild); make module install, make reinstall: use to get kernel back in place

trap is necessary because the process of elevating control

1/17

Wednesday, January 17, 2024 8:10 AM

-First physical 1 GB is filled with kernel code
-The memory map is like an Excel sheet



What is a process – Memory View

Virtual Addresses == Physical Addresses

Virtual Addresses != Physical Addresses

Virtual Addresses are References into Virtual Memory System

The PROCESS sees the virtual addresses and believes them to be real. Behind the scenes, the kernel and memory mapping hardware in the CPU are on the fly translating PROCESS virtual addresses into references to physical memory. The kernel may also be moving process memory to long term slow storage, bringing it in only if needed. We will cover this extensively in a later unit. For now we'll just note it is going on.

1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
3GB
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer. Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

-Kernel launching processes: systemd is pid 1, the kernel boots this process first, systemd spins up other processes

Boot sequence (called "POST" or "Power On Self Test"):

1. Run diagnostic check for bad memory
2. Run firmware
3. Get to BIOS, choose where to load the code from
4. Last stage of the boot sequence is giving the kernel control

1/19

Friday, January 19, 2024 8:04 AM

-A process = a program in execution
-The process has a process counter to know where it is
-Linux memory map:

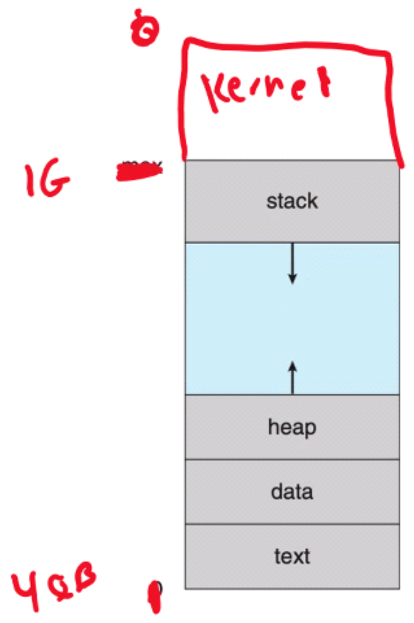
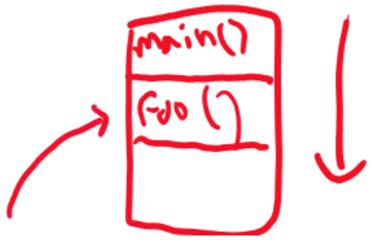
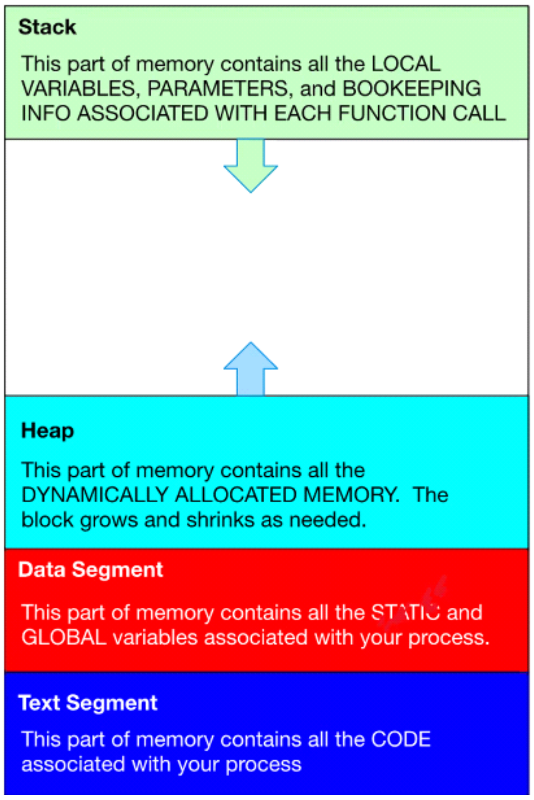


Figure 3.1 Layout of a process in memory.

-Kernel code and trap, interrupt, and fault tables in addresses 0 to 1GB, trying to access code above the 1GB address will throw a fault because that's protected memory
-4GB is the end of the memory map, a 32-bit processor has x4 memory maps for each bus
-Everything from the stack section to the text section is dynamic and virtual memory (like an Excel spreadsheet - you don't know you have 35 thousand cells you can use until you use all 35 thousand of them), addresses 1GB to 4GB grow dynamically to include stuff
-The text session is where our executable code is
-The data section is where our static variables go (including global variables), they are persistent outside the scope of all functions in the program
-The text section and data section are both fixed in size because our memory map knows how big these will be when we run our program (we know how big the program is and how many static variables there are)
-Heap section is where things allocated on the heap go
-The stack is a stack frame with main() at the bottom of the stack and the most recent function call at the top, we pop each record of these function calls until we get back to main:



-The 4 sections of the memory map broken down in code:



Stack

This part of memory contains all the LOCAL VARIABLES, PARAMETERS, and BOOKEEPING INFO ASSOCIATED WITH EACH FUNCTION CALL

Heap

This part of memory contains all the DYNAMICALLY ALLOCATED MEMORY. The block grows and shrinks as needed.

Data Segment

This part of memory contains all the STATIC and GLOBAL variables associated with your process.

Text Segment

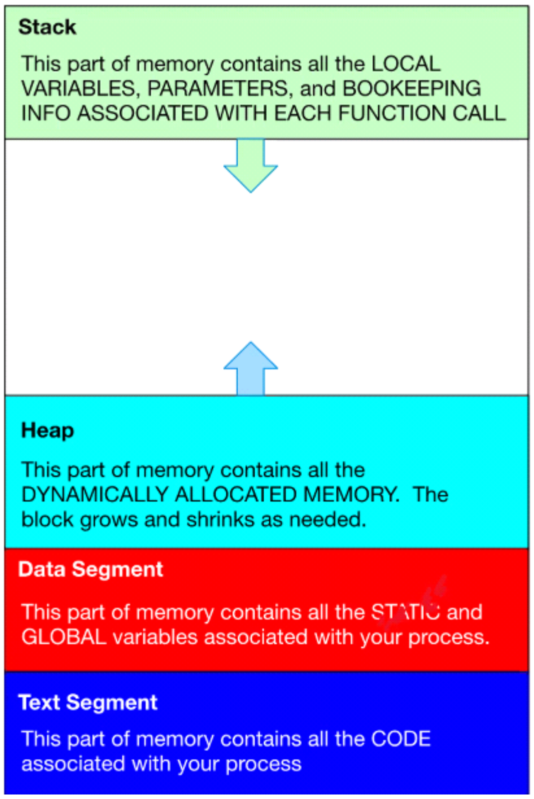
This part of memory contains all the CODE associated with your process

```
#include <stdio.h>
#include <stdlib.h>

int global_var_1;
int global_var_2;

int subroutine(int a, int b)
 { double c;
   static int d = 10;
   if (a > d) d = a;
 }

main()
 { int a;
   int *b;

   // The POINTER VARIABLE b is in the stack.
   // The memory that b points to was "allocated" at
   // runtime by a call to malloc().  malloc() gets memory
   // in the HEAP.  So, the POINTER is in the stack, and
   // memory it points to was requested from, and is in,
   // the heap.

   b = (int *)malloc(10 * sizeof(int));
 }
```

1/22

Sunday, January 21, 2024 3:52 PM

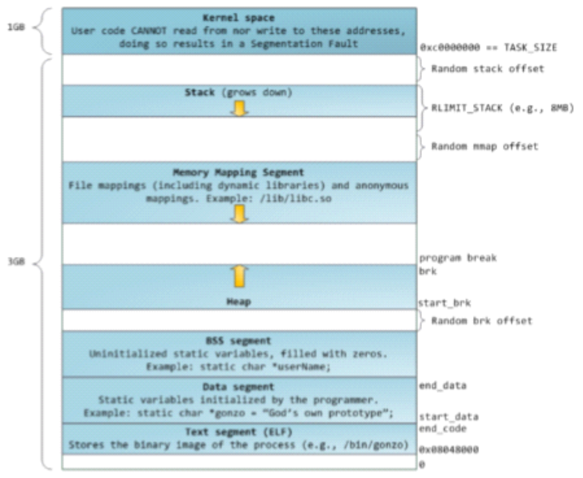
-Recursion works by a different context record of the same code being saved for each recursive function call
-A static variable within the scope of a subroutine has persistence (it's in the .data section of the memory map) but is only active by its name when within the subroutine (i.e. static int d is persistent, it always has the same value even if we leave the subroutine call and come back, say by recursion, but when can't reference the variable d by name unless we're in the subroutine)


Stack
This part of memory contains all the LOCAL VARIABLES, PARAMETERS, and BOOKEEPING INFO ASSOCIATED WITH EACH FUNCTION CALL
Heap
This part of memory contains all the DYNAMICALLY ALLOCATED MEMORY. The block grows and shrinks as needed.
Data Segment
This part of memory contains all the STATIC and GLOBAL variables associated with your process.
Text Segment
This part of memory contains all the CODE associated with your process

```
#include <stdio.h>
#include <stdlib.h>

int global_var_1;
int global_var_2;

int subroutine(int a, int b)
 { double c;
   static int d = 10;
   if (a > d) d = a;
 }

main()
 { int a;
   int *b;

   // The POINTER VARIABLE b is in the stack.
   // The memory that b points to was "allocated" at
   // runtime by a call to malloc().  malloc() gets memory
   // in the HEAP.  So, the POINTER is in the stack, and
   // memory it points to was requested from, and is in,
   // the heap.

   b = (int *)malloc(10 * sizeof(int));
 }
```

-Memory leaks = bad
-Tail recursion is a more safe growth of the stack

A 32-bit Linux process memory map


1GB
3GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;
Data segment
Static variables initialized by the programmer. Example: static char *gonzo = "God's own prototype";
end_data
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

-The .data segment of the memory map has two parts that are unioned together: the BSS segment (uninitialized static variables) and the data segment (initialized static variables)
-Memory mapping segment is like an "island" in the middle of the 3 GB of memory after the kernel

space where we can put shared libraries that we #include<> (these are our DLL files), we don't want to compile the same libraries every time

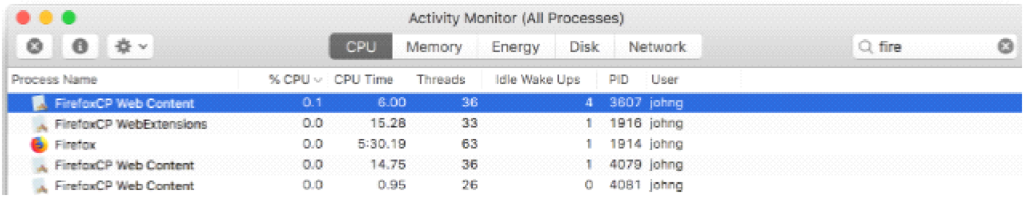
-Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences (there's no conflict between them)

1/24

Wednesday, January 24, 2024 8:06 AM

-We suspend and unsuspend processes all the time
-Processes don't interfere with each other in memory and can flip between each other
-Multiprocess vs. multithreading

-Firefox is parallel, there's one process managing the Firefox window and then several helpers (Firefox has 4 default processors to help)



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
FirefoxCP Web Content	0.1	6.00	36	4	3607	johng
FirefoxCP WebExtensions	0.0	15.28	33	1	1916	johng
Firefox	0.0	5:30.19	63	1	1914	johng
FirefoxCP Web Content	0.0	14.75	36	1	4079	johng
FirefoxCP Web Content	0.0	0.95	26	0	4081	johng

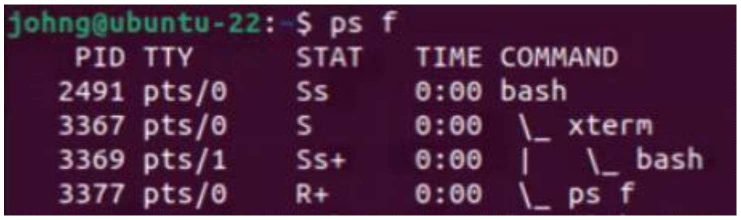
-This is an exploitation of the process model because our processes are actually interacting with each other

1/26

Friday, January 26, 2024 8:03 AM

-systemd is responsible for starting up the "first generation" of child processes
-a process with a "d" at the end of it in Unix is a daemon (i.e. "logind" or "sshd"); a daemon is a process that just waits around until somebody needs it, and then it does something (like how logging into logind spawns children such as the bash process)
-use the "ps" command to get "process status"

  -ps gives us the process tree of all the processes relative to where we are in the process tree: so the bash program running where we called "ps" in the terminal and "ps" itself
  -"ps f" gives us a tree-like print out of our processes
  -it looks like this:



```
johng@ubuntu-22:~$ ps f
    PID TTY      STAT   TIME COMMAND
   2491 pts/0    Ss     0:00 bash
   3367 pts/0    S      0:00  \_ xterm
   3369 pts/1    Ss+    0:00  |   \_ bash
   3377 pts/0    R+     0:00  \_ ps f
```

-the kernel is the government: it picks and chooses which processes get to use the processor and which get to wait
-the process table matches the "process control block" or "context block" which is a thing that bookkeeps the process, to the process ID (PID); process control block -> PID via process table
-the process control block (PCB): holds the priority for certain processes, holds the state for processes (running, stopped, etc.; there's 5 states we care about from the textbook), the process ID (PID) is held in the PCB in addition to the specific PID's parent PID and children PIDs
-Process states (these are universal across Unix and Linux):

  -New: the process is made but has never been assigned a CPU yet, it's a transitory state (a process can only be new once, it cannot become new again)
  -Running: we can run as many processes as we have CPU cores, this is a process in execution
  -Waiting: since many processes are I/O bound instead of CPU bound (i.e. the bash process is waiting around if you're not interacting with it, but once you start sending commands to the terminal, the bash process wakes up), we do this to not be using the CPU on a waiting process; many processes can be waiting
  -Ready
  -Terminated: the process has finished execution but it has not yet been freed in memory by the kernel; the entry in the process table is still there and the PCB has information on a terminated process so that needs to be cleaned, but the process is terminated so it will never execute again; we don't terminate a process immediately because it might still need to give information to a child, so it hangs around for the child process to finish

    -Zombie: is a subcategory of terminated, it's when a child is still running but a parent terminates so the child becomes a zombie and cannot pass the signal on, we need to reap the zombie child process

1/29

Monday, January 29, 2024 8:05 AM

When one process creates another, there are two execution options:

1. The parent continues to execute concurrently with the children
2. The parent waits until some or all of its children have terminated (like the Unix shell waiting for a foreground process to finish)

When one process creates another, there are two memory space options:

1. The child is (at least initially) a clone of the parent (same memory map); it has copies of the same .data (statics and globals) and .text (running code) as the parent
2. The child process has a new program loaded into it and "fresh" data

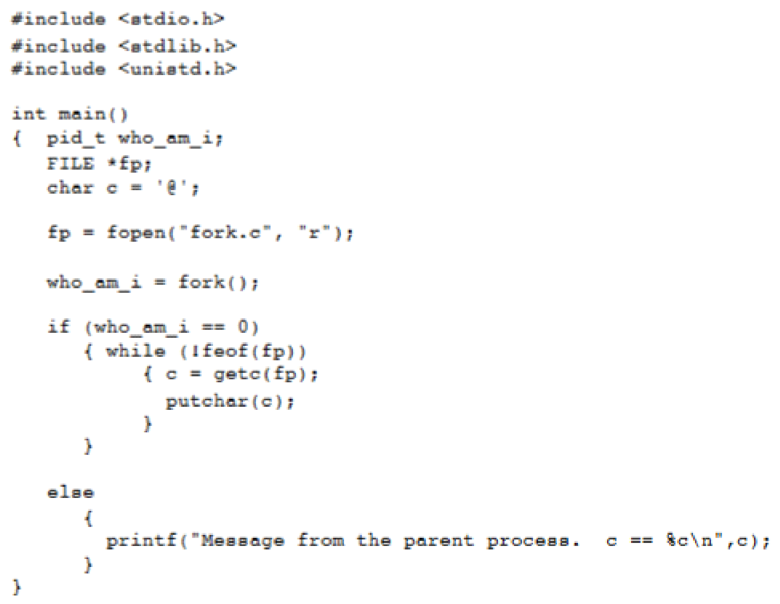
How Unix creates processes:

1. The process makes a clone of itself; the clone contains all the .data and .text (code) of the original but in a different place in memory; then they can diverge from each other from there and the clone can access anything the original could (they're disjoint)
2. We don't have to pass information to the clone, the clone should already have everything

-When we call fork(), the clone as at the same point in its execution cycle as well as the same memory map
-We can use kernel calls to get our child's parent's PID (since the child's PID is 0); i.e. call get_pid()
-You have to keep track of the child's PID, that's on you
-A fork() bomb is when you keep forking and creating more processes

1/31

Wednesday, January 31, 2024 8:03 AM

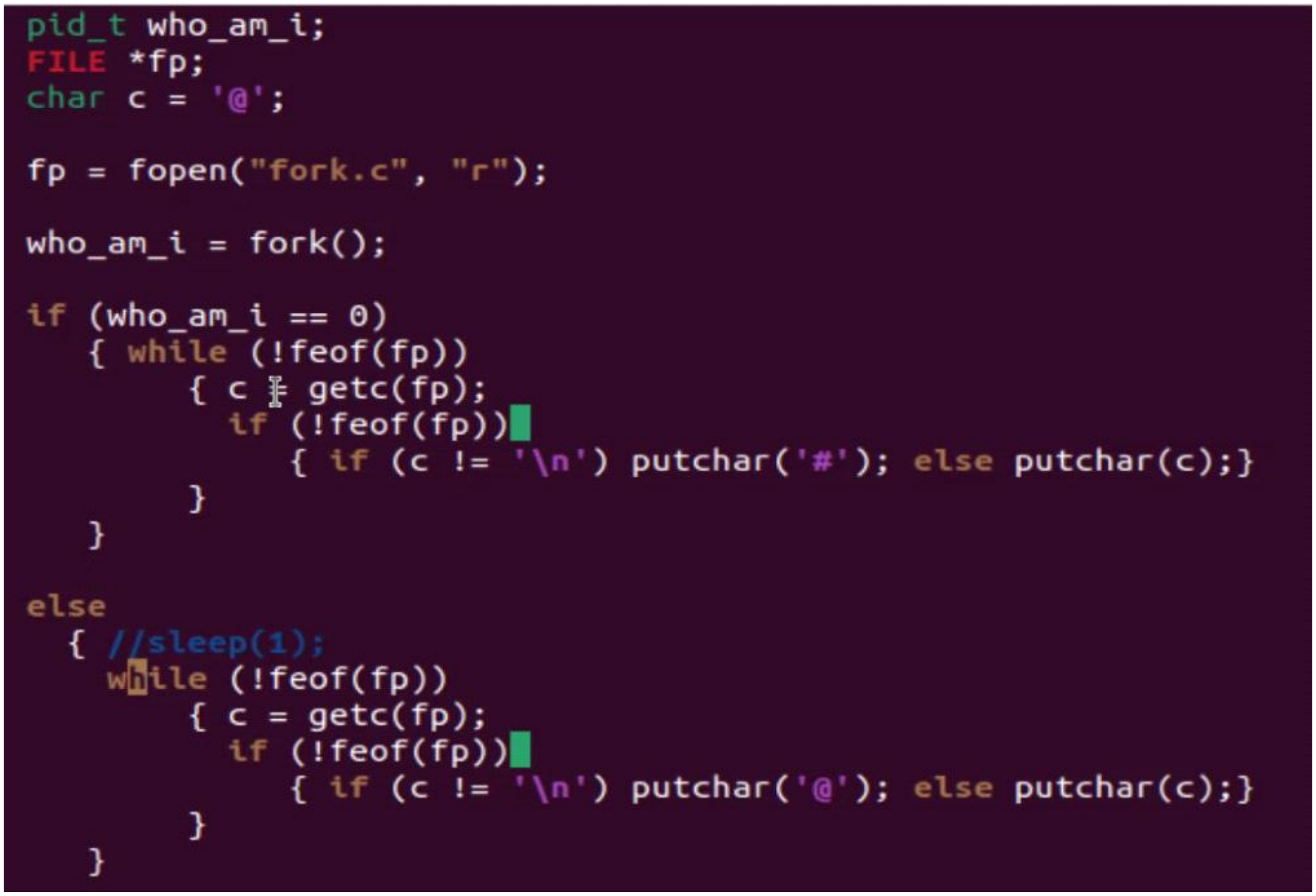


```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{  pid_t who_am_i;
   FILE *fp;
   char c = '@';

   fp = fopen("fork.c", "r");

   who_am_i = fork();

   if (who_am_i == 0)
      { while (!feof(fp))
           { c = getc(fp);
             putchar(c);
           }
      }

   else
      {
        printf("Message from the parent process.  c == %c\n",c);
      }
}
```

-This is our first example of a shared resource: fopen() keeps track of two of the same resources: the open file we're reading from

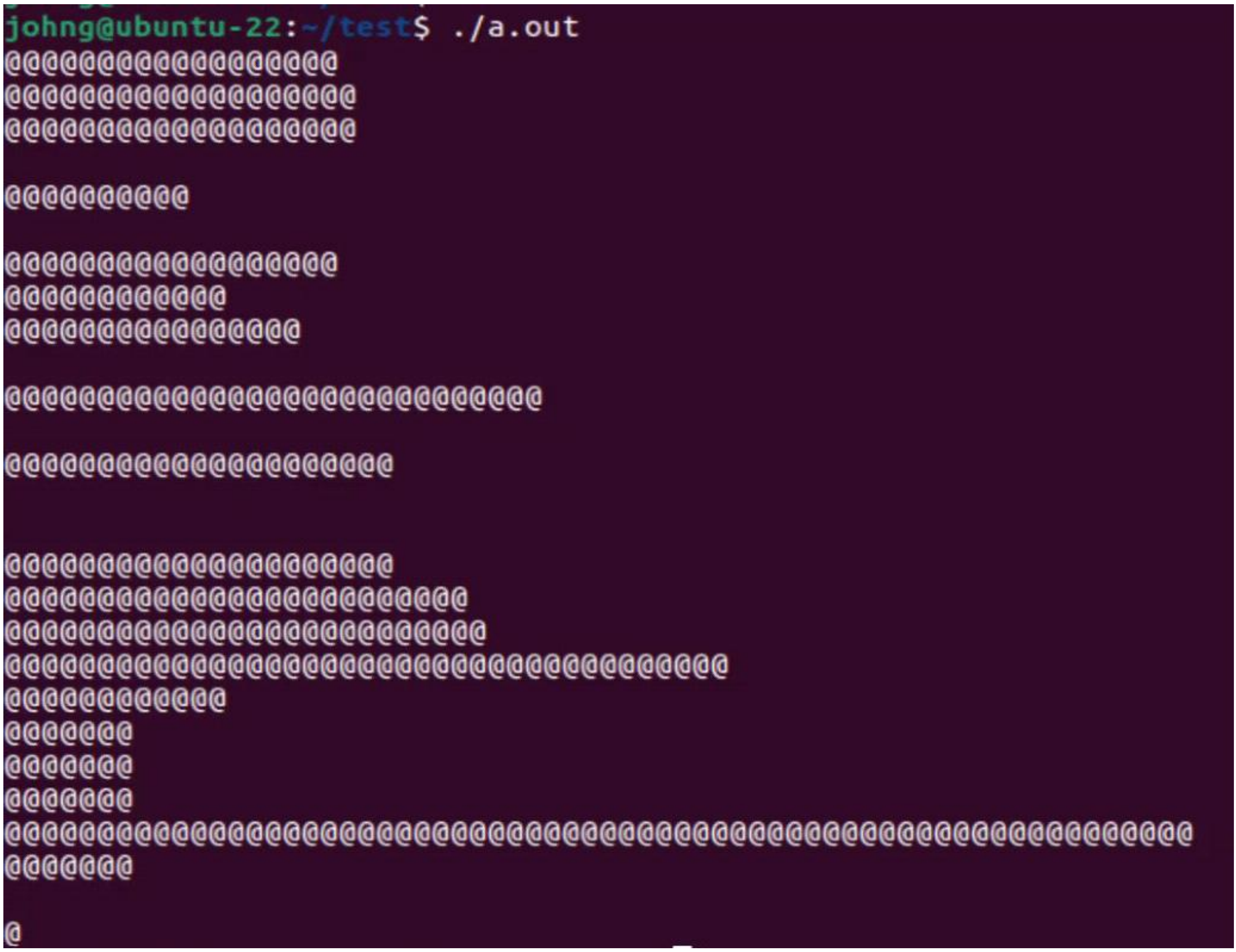
2/2

Friday, February 2, 2024 8:04 AM



```c
pid_t who_am_i;
FILE *fp;
char c = '@';

fp = fopen("fork.c", "r");

who_am_i = fork();

if (who_am_i == 0)
   { while (!feof(fp))
          { c = getc(fp);
            if (!feof(fp))
               { if (c != '\n') putchar('#'); else putchar(c);}
          }
   }

else
  { //sleep(1);
    while (!feof(fp))
          { c = getc(fp);
            if (!feof(fp))
               { if (c != '\n') putchar('@'); else putchar(c);}
          }
   }
```

The output of this code is this:



```
johng@ubuntu-22:~/test$ ./a.out
@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@

@@@@@@@@@@

@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@
@@@@@@@@@@@@@@@@

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

@@@@@@@@@@@@@@@@@@@@@


@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@
@@@@@@@
@@@@@@@
@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@

@
```

-If we uncomment the sleep(1) on the parent process, we get all '#' chars from the child process reading from the file

-We don't know a priori what the pattern is for which will be first with the scheduler, it runs stochastically fighting over a shared resource (the file we're fopen()ing)
-If both ran at the same time, it would be a mix of #'s (from the child) and @'s (from the parent)

Communication among process:
- -Interprocess signaling; pipes, sockets, and files
- -Shared memory models (threads), messaging models

2/5

Monday, February 5, 2024 8:10 AM

-Shared memory = some memory that is physically in the same place as both process A and process B (a segment of the memory is in process A and another segment of the memory is in process B).

2/7

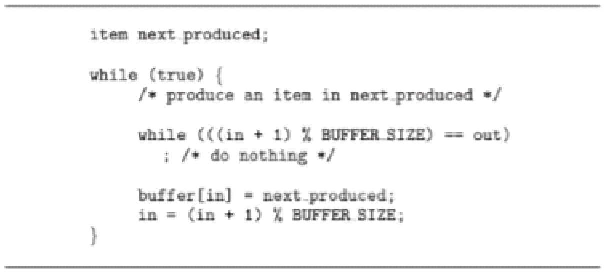
Wednesday, February 7, 2024 8:07 AM

-Pipes are like postcards, once you send it, it's the post offices problem (where the post office is the kernel)
-Signals between processes are a messaging model
-A classic pipe in Unix was that chars were pushed through
-A pipe is file-like object, it exists both within a parent and a child
-Shutting down an end on the pipe makes a one-way communication
-Orphan processes are taken care of by being "adopted" (for lack of a better word) by the kernel at the systemd process
    -An orphan process can temporarily be a zombie before being adopted

-Shared memory: imagine creating a global variable that is not copied over by fork() but is actually the same physical memory shared between two processes; this is dangerous because of race conditions

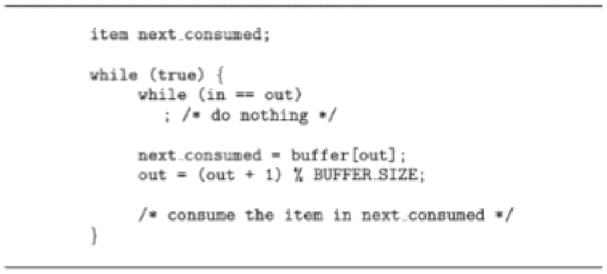
-What are the cost and benefits of using a messaging model?
    -Cost: overhead - sending data from process A to process B requires preprocessing, putting the data somewhere in the kernel space which takes up additional space, we're using the kernel like a trucking company
    -Benefit: subtle issues of timing are being handled by the kernel, a package delivered at 4 AM isn't your problem, it's the delivery company's problem
    -A message passing model makes sense for significantly rare events/updates
-What are the cost and benefits of using a shared memory model?
    -Cost: having several threads talking to the same memory means you don't want the streams to cross
    -Benefit: there's no overhead moving the data, you already have the shared memory readily available to access
    -If two processes are constantly updating, you don't want to pay the overhead, so you use a shared memory model, but now you're in charge of synchronization

Producer Consumer Model



```
item next_produced;

while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Figure 3.12 The producer process using shared memory.



```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

Figure 3.13 The consumer process using shared memory.

`in`	is a SHARED variable in both processes
`out`	is a SHARED variable in both processes
`buffer`	is a SHARED variable pointer that points to a block of memory also shared by both processes. The buffer has `BUFFER_SIZE` slots in it

-This is done through a circular queue

-Technically a Unix pipe is a producer-consumer, except the queue is handled by the kernel

2/14

Wednesday, February 14, 2024 8:14 AM

-fscanf(stdin) and fprintf(stdout) or getchar() and putchar()

2/12

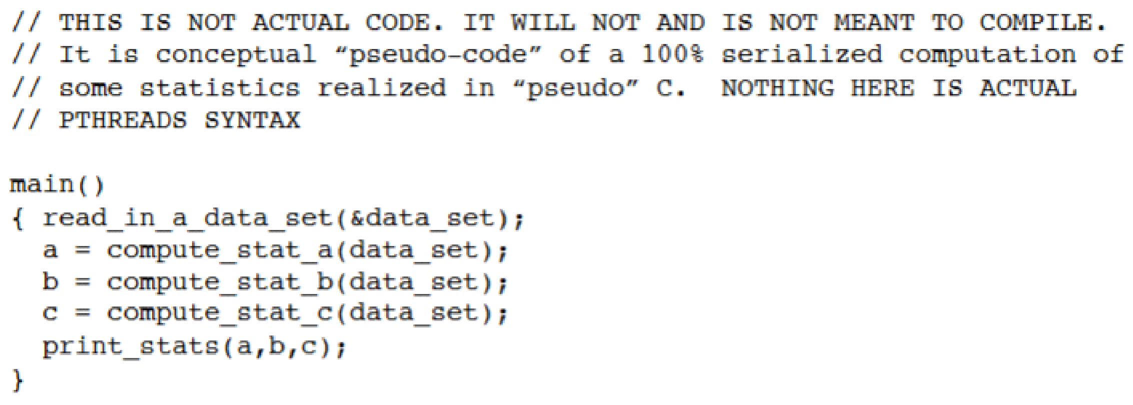
Monday, February 12, 2024 8:07 AM

-No simultaneous access of data but involves multithreading for next homework, it's just an exercise of making calls to the pthreads C library, this is good because we don't have to deal with resources touching each other
-All threads have the same:
- -.text segment (they have the same running code)
- -registers
- -run counters
- -files (a file open in one thread is open in all of them)
- -.data segment (they have the same statics and globals, they physically share the same memory between global variables)
- -individual stack sections

-Fun idea: point a thread to the stack segment of another thread
-Lightweight processes - another name for threads, the scheduler treats the "ready" state of a process as a priority queue; it's very efficient to put a thread to sleep (context switching to them) when the scheduler treats them akin to individual processes
-Parallelism and concurrency are our two tools of propping of multithreaded programming
- -If you have a single core and you have multiple processes, you'd use concurrency because you're changing processes so fast, it's like a flipbook

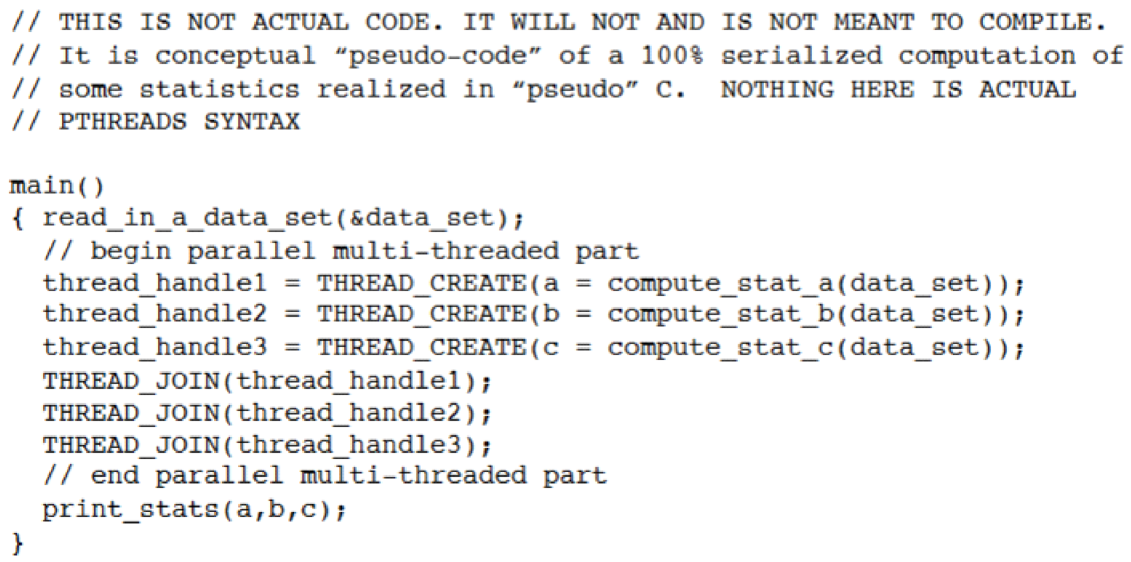
-Process model == no threads; our newer memory model has threads
-Sharing globals between threads is super fast (even faster than Unix pipes), the advantage of the fast contact is huge but the downside is that we have to be careful because there's no debugger for multithreaded code (the debugger is nondeterministic)
-Joinable threads - we start the child process and we wait for it, the entity that launched the thread is the thing that's waiting
- -Dumb question - what if a thread makes more threads?
- -Only one entity should be launching the threads, the threads themselves should NOT be launching threads off of themselves, but the entity launching the threads can launch as many as it needs

-Asynchronous threads -



```
// THIS IS NOT ACTUAL CODE. IT WILL NOT AND IS NOT MEANT TO COMPILE.
// It is conceptual "pseudo-code" of a 100% serialized computation of
// some statistics realized in "pseudo" C.  NOTHING HERE IS ACTUAL
// PTHREADS SYNTAX

main()
{ read_in_a_data_set(&data_set);
  a = compute_stat_a(data_set);
  b = compute_stat_b(data_set);
  c = compute_stat_c(data_set);
  print_stats(a,b,c);
}
```

- -Important observation: the order of execution and any sense of parallelism doesn't matter if we were running tasks that don't modify any data (i.e. a database), they might as well be running at different times



```
// THIS IS NOT ACTUAL CODE. IT WILL NOT AND IS NOT MEANT TO COMPILE.
// It is conceptual “pseudo-code” of a 100% serialized computation of
// some statistics realized in “pseudo” C.  NOTHING HERE IS ACTUAL
// PTHREADS SYNTAX

main()
{ read_in_a_data_set(&data_set);
  // begin parallel multi-threaded part
  thread_handle1 = THREAD_CREATE(a = compute_stat_a(data_set));
  thread_handle2 = THREAD_CREATE(b = compute_stat_b(data_set));
  thread_handle3 = THREAD_CREATE(c = compute_stat_c(data_set));
  THREAD_JOIN(thread_handle1);
  THREAD_JOIN(thread_handle2);
  THREAD_JOIN(thread_handle3);
  // end parallel multi-threaded part
  print_stats(a,b,c);
}
```

-THREAD_JOIN is like a wait function but for threads (that act like children)
-Threads are individually schedulable by the kernel so you have no idea how the kernel will schedule them
-"It's never the same river twice", conditions will almost always be different

-POSIX Pthreads have an equivalent in Windows with Windows Threads API

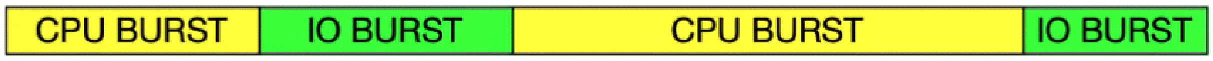
-If ANY thread in a process runs exit() or _exit() -- this stops the whole process (regular exit() does a clean exit, _exit() stops the termination but leaves all the file descriptors and file objects open, child process should use _exit() because the parents may need its resources) and also all the threads are destroyed

2/16

Thursday, February 15, 2024 11:31 PM

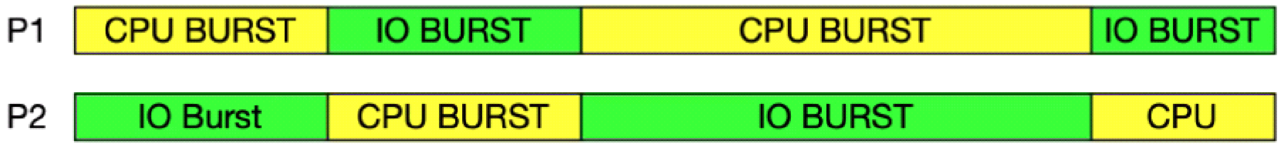
-Most processes are in some kind of wait state or ready state, very few are actually being executed by the CPU

-If we're running one process, we go between the CPU and not running it on the CPU:



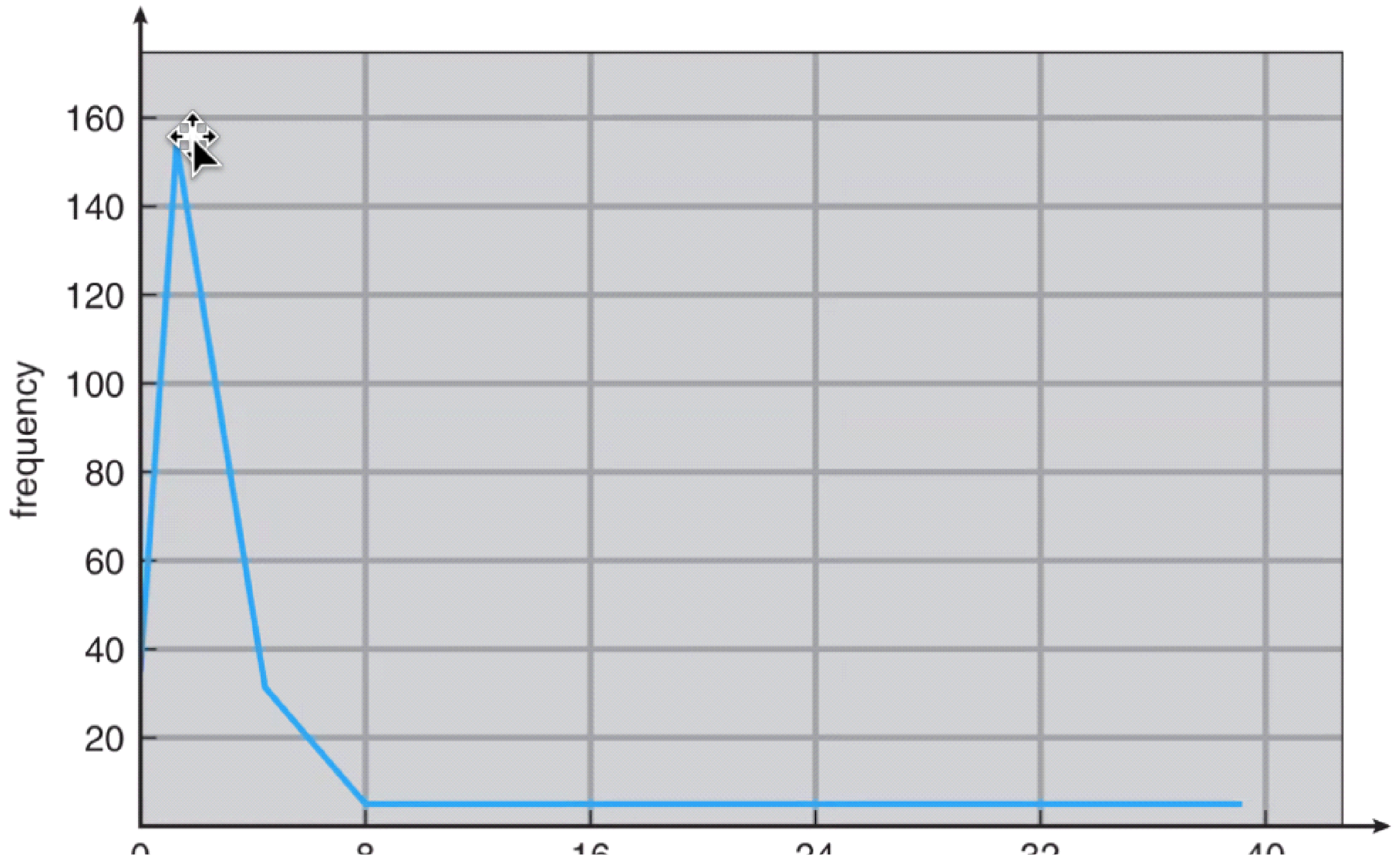
-In a perfect model: a process during a CPU burst is in a ready or running state, a process during an IO burst is in a waiting state

-
Parallel means being run by two cores at the same time, concurrent is context switching between the sa me process
-If we're running two processes, we switch between using the CPU (concurrency); this is the perfect world where we waste no CPU time



-The process scheduler's job is to get us to as close a result as the model above where multiple processes line up and we're not wasting CPU time
-The chaos factor comes in where each process might not be ready to need CPU time, when CPU time is available; that is, we're hoping a CPU burst is a ready or running state and an IO burst is a waiting state, this is **not** always the case, we have hundreds of processes running in parallel

-On average, CPU burst time looks log-normal



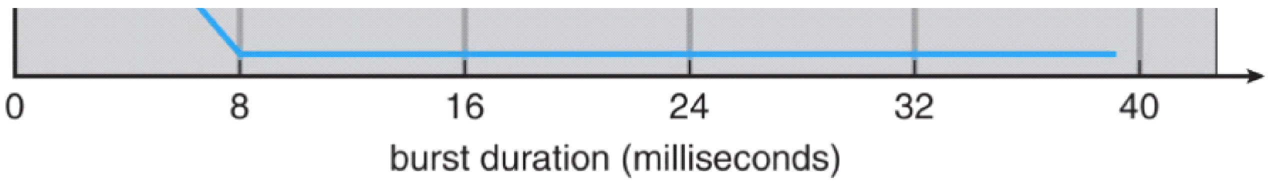


Figure 6.2 Histogram of CPU-burst durations.

- -This graph means we're guessing correctly most of the time
- -One way to schedule is to make a model based on the average CPU bursts of every process, find out the number, then we run bursts for that long
- -A more advanced scheduler is keeping track of how long average CPU bursts for individual processes (on top of the average CPU burst time of every process), this accounts for some processes taking longer than others because we can adjust

-When we build schedulers, the quintessential condition is asking who gets to use the CPU and when, it will never be a perfect system either because you might have different priorities
-The Linux kernel out-of-the-box has a process scheduler optimized for a generic user, a gamer might want to optimize the process scheduler differently (i.e. reduce lag)
-All processes have IO bursts and all have CPU bursts, it just differs in the proportion - GUIs are "IO bound processes" because the IO bursts are longer than CPU bursts, computing-intensive processes are "CPU bound processes"

2/19

Friday, February 16, 2024 8:57 AM

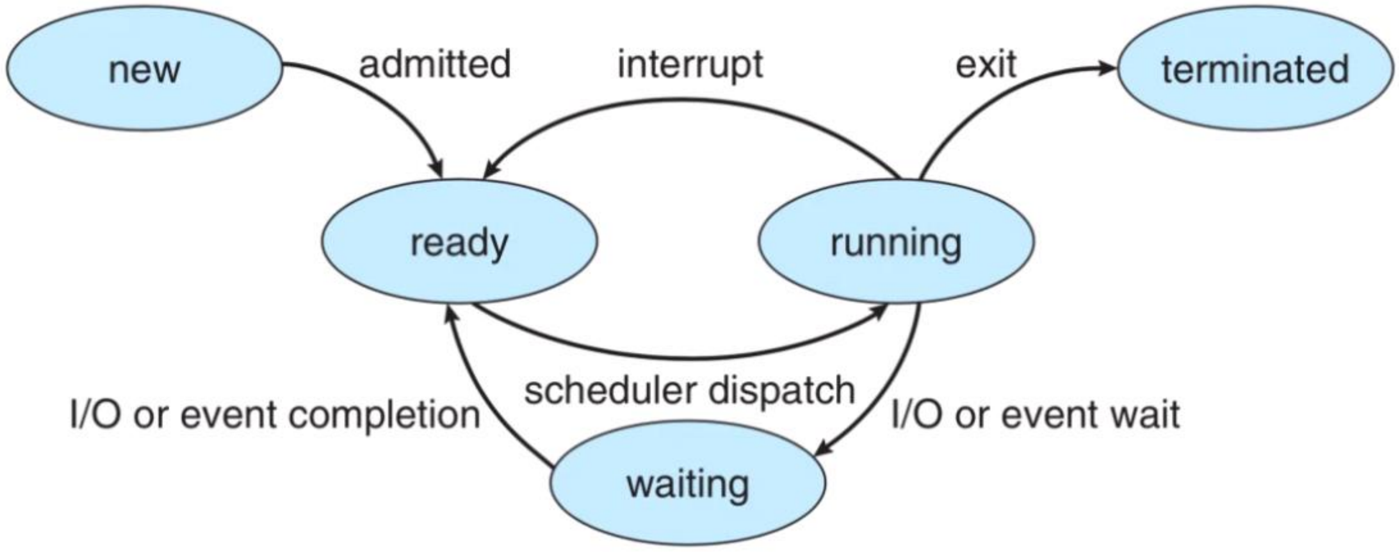


Figure 3.2 Diagram of process state.

-Ready means a process is ready to be run, but is not being run
-The scheduler (via a scheduler dispatch) schedules a ready process
-When the scheduler schedules a ready process, its state is now a running process
-Terminated means the child still needs to have some data sent but is waiting for the kernel to deal with it so it can be released
-Wait state means we're in an IO burst, we're waiting for something else, even if there's a job for us to do (analogy: you're doing construction and you forgot your tools so you have to go back to get them, you can't work until you have your tools)
-We can go from running to ready through an interrupt so a process can stop from hogging all the CPU time (the kernel might push the process off for taking too much time; this used to not be a thing and it was called a "non-preemptive kernel")
-Each of these states has a queue: FIFO queue (new and terminated states), priority queue (ready state) to prevent "process starvation" (each process gets attention),
-Scheduler is run 10's of thousands of times in a given minute, the code needs to be precise or you pay an amortized (overhead) cost for each context switch - minimal memory and minimal time

2/21

Wednesday, February 21, 2024 8:11 AM

CPU scheduler definition:
- Decides when the process gets to get off the CPU
- The CPU scheduler decides when and how process control blocks (PCBs) move from queue to queue

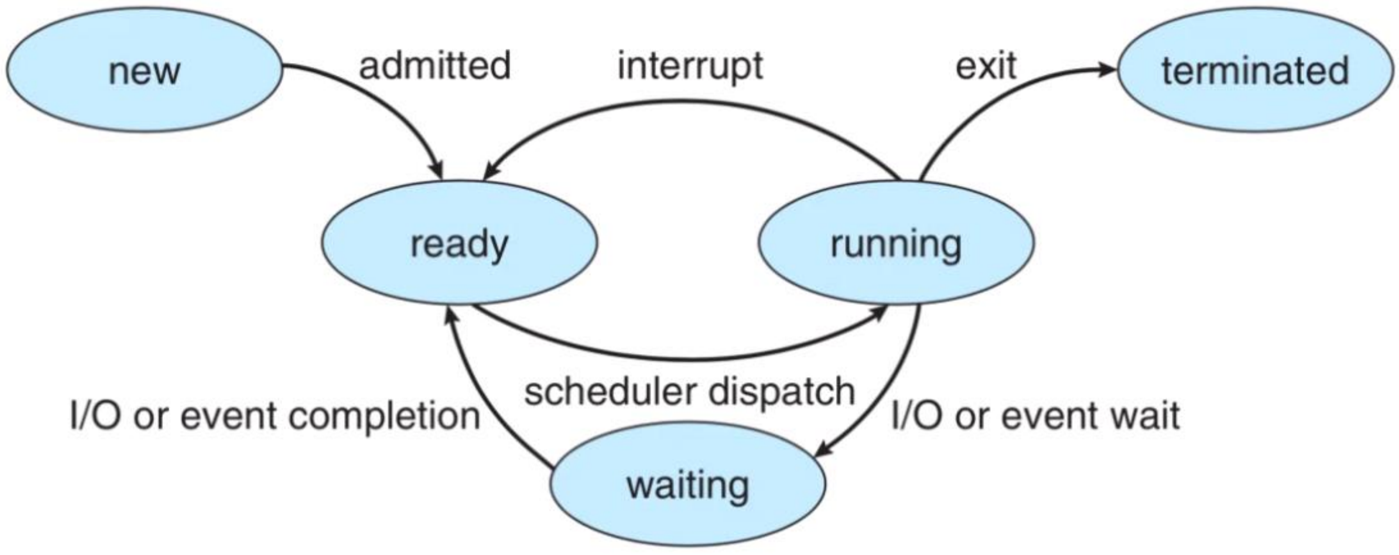


Figure 3.2 Diagram of process state.

CPU scheduling events, CPU scheduling takes place under these circumstances:
1. When a process switches from running to waiting as the result of an IO request or as the result of waiting on a child process to end;
2. When a process switches from running to ready (e.g. an interrupt); the process has been on the CPU too long so the kernel boots it off
3. When a process switches from waiting to ready (e.g. completion of an IO event); this is a unique event that occurs based on priority of certain processes (some process joined the queue that is very important)
4. When a process terminates

If #1, #3, and #4 only can occur on the process scheduler, then our process scheduler is a *nonpreemptive* or *cooperative* scheduler

If #2 can exist, then the process scheduler is a *preemptive* scheduler

Two rules:
1. When a CPU becomes free, run the scheduler
2. When a new process enters the ready queue, run the scheduler

The dispatcher
- The dispatcher is the kernel module that gives control of a CPU to a specific process
- It handles context switches and transfers control to the proper location in the user process
- The dispatcher must be fast, if it's not, it has "dispatch latency"
- Note from before: context switching a thread (lightweight process) is much faster than a process (we have to update memory maps and virtual memory)

Our scheduling criteria, an analogy is "how happy, in mass, is everyone waiting in line at an amusement park"
Long-term aggregate statistics of how happy our processes are

Pareto optimality, we can't optimize two things to maximum at the same time:
https://en.wikipedia.org/wiki/Pareto_efficiency

-Unit of work = CPU burst, not whole length of time (lifespan) of a process

Scheduling criteria we want to maximize:

1. CPU utilization - the amount of time that a CPU is actually doing work; 100% CPU utilization means we're always running processes; this is several hundred percent in modern Linux systems
2. Throughput - the number of processes that are completed per unit time, the measure of the rate that we're pushing work through the system; measures how much work we're getting done (i.e. 3 processes a second)

-Can find these using the "top" command in Linux

Scheduling criteria we want to minimize:

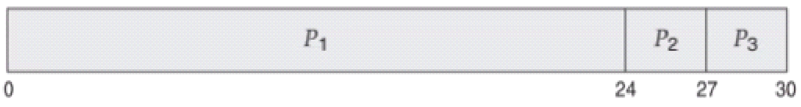
1. Turnaround time - the amount of time it takes from process submission to the new queue to its exit from the termination queue; the time of when you're born to when you're dead (as a process); measures how long it takes to get a single job done
2. Waiting time - the amount of time a process spends in the wait queue; measures how good a kernel is at moving work to the CPU
3. Response time - the amount of time it takes for the first IO to occur; this factors out the IO times that turnaround time counts

2/23

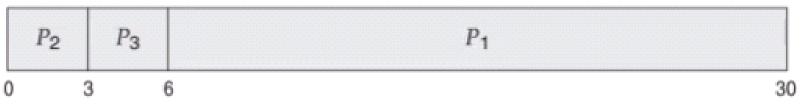
Friday, February 23, 2024 8:02 AM

-Scheduling algorithms:
-First come, first serve (FIFO) - first process ready is the first process running

If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process P_1, 24 milliseconds for process P_2, and 27 milliseconds for process P_3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P_2, P_3, P_1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

-Average times will be very long in first come, first serve (and you can't control when a process comes around); here are the wait times of the processes above:

P3 = 0

P2 = 3

P1 = 6

Avg = 3

51
.3

P1 = 0

P3 = 24

P1 = 27

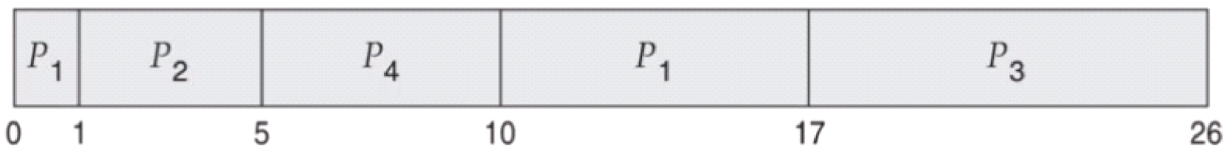
P3 = 21

P1 = 27

-Does not have fairness (fairness is when not all processes get a fair shot to run) and has process starvation

-Analogy: fast pass at amusement park might not be fair but it prevents starvation if everyone is riding

-Shortest job first (SJF):



-Shortest job first (SJF) is technically a priority queue, but its priority is based on job shortness

-We calculate CPU burst times by some lookup table of the last 10 runtimes that we can keep statistics on

-Recurrence relation for running average of observations (also used in neural networks), for estimator calculating when the next process should be:

$$\tau_{n+1} = \alpha \ t_n + (1 - \alpha)\tau_n.$$

$$\tau_{n+1} = \alpha t_n + (1 \ - \ \alpha)\alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} \tau_0.$$

-Shortest job first (SJF) is optimal for reducing average wait times

-The downside is, we have a higher probability of process starvation (processes with smaller request keep "cutting the line" in front of larger processes, you might not even schedule a large process if you keep getting many smaller processes)

-NOTE: nobody uses this type of scheduler really

2/26

Monday, February 26, 2024 8:06 AM

-Recurrence relation for running average of observations (also used in neural networks), for estimator calculating when the next process should be:

$$\tau_{n+1} = \alpha \ t_n + (1 - \alpha)\tau_n.$$

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0.$$

-The estimate of the future CPU burst is going to be tau_(n+1):
- -The previous burst is…
- -Tau_n is the estimated CPU burst time
- -T_n is the actual CPU burst time
- -Alpha is how quickly or slowly we want the exponent to decay (we can control alpha); between 0 and 1
- -Ultimately, we like this equation because it's cheap to compute (2 multiplications, 2 additions)
- -This cannot have process starvation

-Priority scheduling
- -Each process gets a "priority score" - it comes off the wait queue and into the ready queue in a priority order
- -Easy statistics: age of process on queue, user specified privilege level

-Round-robin scheduling (RR)
- -Implement with a circular queue
- -RR is a FCFS/FIFO-like (first come, first serve / first in, first out) scheduler with preemption
- -It's like a carousal where you get pulled off and put onto a CPU with a budgeted time
- -Each process that moves to a CPU may keep the CPU for its whole CPU burst OR a set amount of time, whichever happens first
- -As soon as a process surrenders or gets kicked off the CPU, it goes back to the wait queue
- -In order for RR to starve, we need to have many processes entering the circular queue

2/28

Sunday, March 3, 2024 4:00 PM

== crossing the processes ==

> producer consumer
- busy wait will come back to bite us
- buffer is effectively a circular array
- we don't know the relative timings, we don't know the relative CPU and IO bursts, we don't know what the scheduler is doing
- how many assembly language instructions is "count++" and how many is "count--"? you don't know, (ideally 1) because there are so many variances between compilers and hardwares that anything is really possible
- RISC processors generally don't do calculations on memory
- saying "we expect" because it makes sense to us is dangerous
- producer and consumer are accessing identical memory
- the increment operation is not atomic on the machine level
- you don't know if or when instructions will interleave, but they are pretty rare
- this produces the worst kind of error: quietly producing a wrong value and passing it along as if nothing happened

# "some people are going to have to go home and think on it, some people are going to have to go home and drink on it"

3/1

Monday, March 4, 2024 8:03 AM

== protections ==

> producer consumer
- when we give control to a scheduler it becomes possible for you to overlay the assembly instructions of two threads in a way that it will execute at the same time (interleaved)
- you can try to make the kernel do all that management, but it will almost certainly impact performance
- you can't allow the micro-exceptions to exist because it gives the worst kind of errors: silent incorrect results

> locking primitives
- system call is "lock", another is "unlock"
- each lock/unlock call is a global variable
- lock checks the variable inside, if it's locked already, lock goes into a wait until the variable becomes unlocked
- unlock takes the variable and unlocks it
- lock is a check and hold or a change and pass
- if the variable is false, lock changes it to true and moves on
- if the variable is true, lock will wait until the variable is set to false by an unlock call
- if the implementation is correct, locking and unlocking MUST be atomic
- if the change wasn't atomic, it couldn't prevent overleaving
- if the producer is running in one thread and the consumer is running in another, one will get to the lock first
- all three instructions of one will run before all three instructions of the other will run
- locks prevent interleaving
- lock is atomic, unlock doesn't need to be
- by allowing the user level programmer access to lock/unlock, the programmer can protect the tiny segments of code where interleaving can happen
- the kernel had no idea where interleaving can and can't happen
- the programmer protects ONLY the danger segments to allow the scheduler to optimize everything else
- you lose optimal scheduling when you lock, but at least you get the right result
- using these tools is like playing cat and mouse: you want to protect as little as possible to maximize optimization, but you can't protect too little or you risk leaving danger sections unprotected
- lock() and unlock() are system calls promised to be atomic

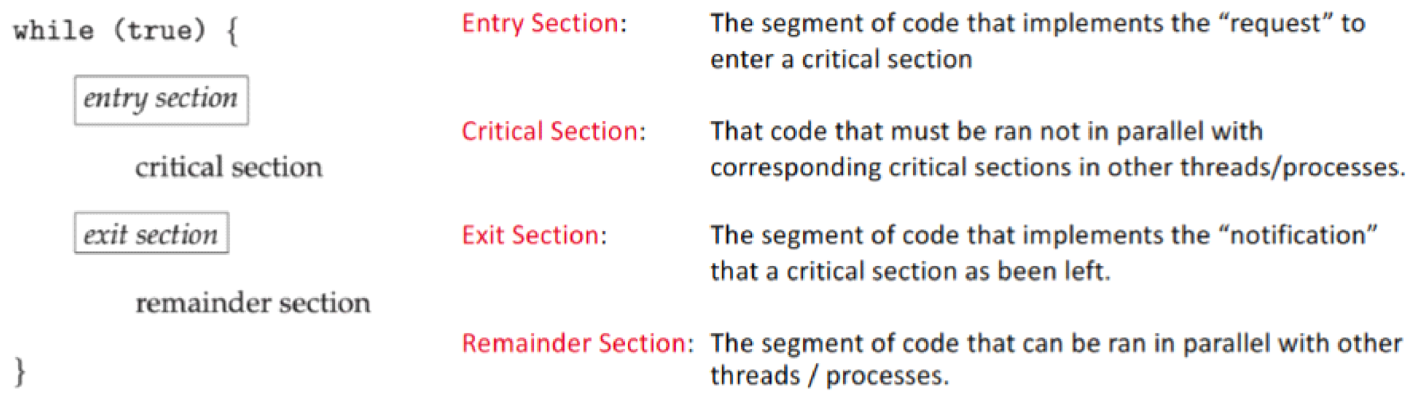
> the critical section problem
- locking primitives solve the critical section problem
- a critical section is a section of code that accesses data that is potentially running in some other process or thread
- "count++" and "count--" are both critical sections in our producer/consumer example
- reading is not a critical section usually
- a bathroom on an airplane is like a critical section, whatever's happening in there is still critical whether there's a lock on it or not, there's just a higher chance of unfortunate incursion events if there's no lock
# CRITICAL SHIT!
- the critical section is the place where the changes happen
- corresponding critical sections are corresponding if they're dealing with the same critical data
- it is possible for one piece of code to have more than one critical section, but those critical sections

may involve different resources
- analogy would be having one lock for EVERY bathroom instead of one lock for bathroom, it still solves the problem but causes serious issues with efficiency
- it is vital that when any process or thread is running in a critical section, no other process or thread can run in its own corresponding critical section
+ entry section: the code that implements the request to enter the critical section
+ critical section: code you MUST protect
+ exit section: the code that implements the notification that someone left the critical section
+ remainder section: code you need not protect
- the remainder is both before and after the critical section
- how do i design code that manages the lock? there are properties that must be honored
- if all three properties are not satisfied, it is not a good lock
+ mutual exclusion: if process P1 is in its critical section, no other process may be in its corresponding critical section
+ progress: if no process is executing in the critical section, only those processes that are not in their remainder sections can participate in who goes into the critical section next (if nobody's in the crit section, only processes in line can decide who goes next), the decision cannot be delayed indefinitely
+ bounded waiting: there is a finite bound on how many times other threads/processes are allowed to enter their critical sections before a process/thread is allowed to enter on its own (gets rid of the cutting in line problem, no starvation)

3/4

Monday, March 4, 2024 8:03 AM

The Critical Section Problem:
-If we have code from another process that's updating running code of a process we're looking at, how does this update code in our current process without overwriting?



```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

Entry Section:	The segment of code that implements the “request” to enter a critical section
Critical Section:	That code that must be ran not in parallel with corresponding critical sections in other threads/processes.
Exit Section:	The segment of code that implements the “notification” that a critical section as been left.
Remainder Section:	The segment of code that can be ran in parallel with other threads / processes.

Solutions to the Critical Section Problem:

1. Mutual exclusion - if process 1 is in its critical section, no other process may be in its critical section
2. Progress - if no process is executing in its critical section, then only those processes NOT in their remainder sections may participate in who goes into the critical section next
3. Bounded waiting - there's a limit how many other threads/processes are allowed to enter a processes critical section before it's allowed to enter its own

-Mutexes are mutual exclusion operators
-Semaphores are more classic
-Monitors don't matter
-A reader-writer lock is a specific kind of mutex, we're doing this using a hardware solution
-Peterson's Solution is like a double deadbolt door, we have two flags that lock and unlock for entry (one for each process)
-You cannot enter the remainder section of your code without in a process without surrendering access to the processes critical section
-Peterson's solution fails because 1.) compiler might intermix our critical section and remainder section and 2.) there's no guarantee that the Booleans are atomic

```
int turn;
boolean flag[2];
```

There are TWO processes/threads. One called P_0 and the other called P_1

Both processes SHARE the variables turn and flag[]

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Both processes run the same code

in either of the two processes, `i` refers to the "PID" of the process that we are examining running the code and `j` refers to the "PID" of the OTHER process in the pair. For a process 0, `i == 0` and `j == 1` and for a process 1, `i == 1` and `j == 0`.

`turn` records whose turn it is to enter the critical section
`flag[x]` keeps track of if process `x` wants to enter its critical section

Mutex locks:
-Mutex locks can be true or false; require/release or lock/unlock (preferred terminology)
-A global variable that acts like a deadbolt, the bookkeeping for mutual exclusion, progress, and bounded waiting is handled for us

3/6

Monday, March 4, 2024 8:55 AM

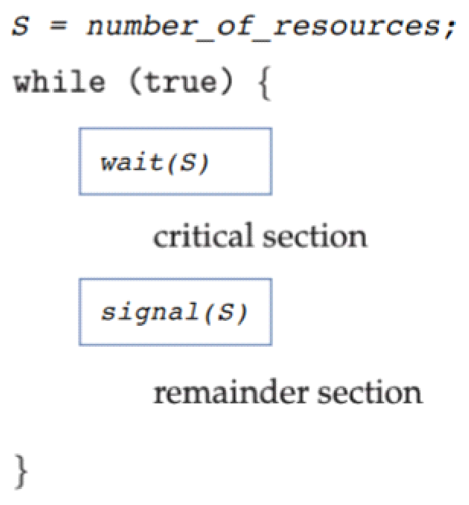
-Mutex lock contention:
- -highly contended locks break your efficiency and can even get rid of any advantage of multiprocessing at all
- -if you're writing code, you should make sure your locks are lowly contended
- -if you have a bunch of threads all waiting on one lock, they're not working
- -generally speaking, you want to avoid algorithms with highly contended locks if possible
- -protecting the entirety of the code in a critical section is a horrible idea

-Spinlock - a spinlock is an implementation where you keep trying the mutex lock (in an infinite while-loop)
- -It's a "busy wait" (you're in the running state, not the wait state, yet you're waiting for the lock to be lifted)

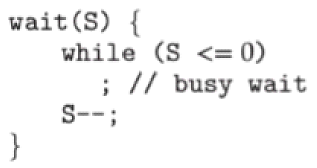
-Schedule-yield (sched_yield() system call) - we put our running process to sleep (entering the wait state) instead of actively using the ready state on a heavily contented lock
- -Downside: we need to do a context switch which has overhead to switch states from running to wait
  - -We actually have to do TWO context switches: one context switch to come off the running state into the wait state when we're put to sleep AND one context switch to re-enter the running state from the wait state

-Spinlock vs. schedule-yield - you should always choose whichever locking mechanism is cheaper
- -Real mutexes are hybrid, they use both of these solutions - we spinlock briefly and if we wait too long, we do a schedule-yield

-Semaphores
- -P for Dutch proberen (try) and V for Dutch verhogen (increment) = wait() and signal()

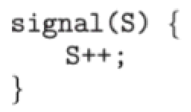


```
S = number_of_resources;
while (true) {
    wait(S)
        critical section
    signal(S)
        remainder section
}
```



```
wait(S) {
   while (S <= 0)
      ; // busy wait
   S--;
}
```

P(S)



```
signal(S) {
   S++;
}
```

V(S)

- -A mutex is a subset of a semaphore, a mutex can fake a semaphore
  - -That "busy wait" inner while-loop is a spinlock

-Deadlocks
- -Idea: a process needs two mutex locks to be able to leave, these mutex locks are shared resources between processes, if there are two processes and one process grabs one mutex lock and the other process grabs another mutex lock, now neither process can leave because the processes need both mutex locks each

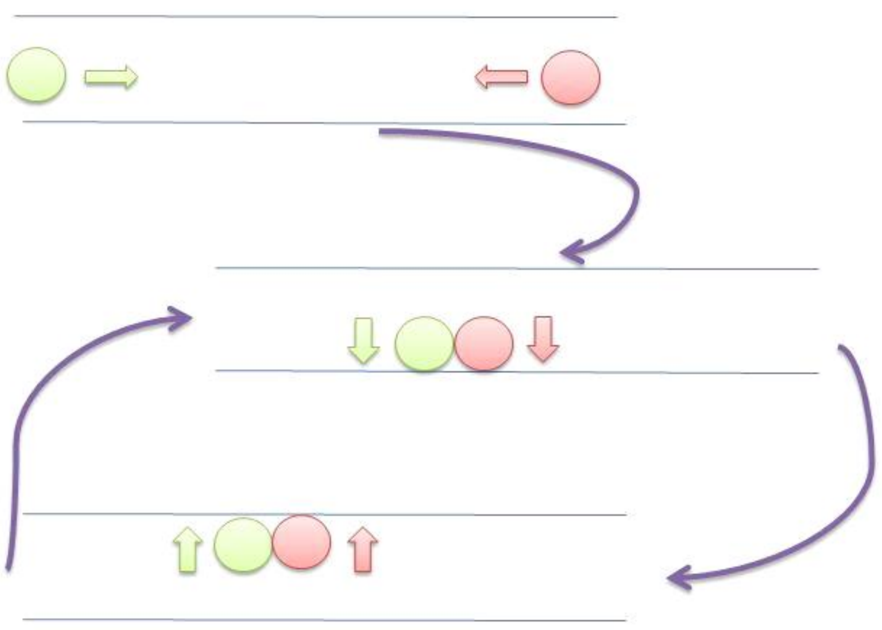
3/18

Monday, March 18, 2024 8:11 AM

Recap:
-"loss of liveness" is a situation that occurs when there are two or more lines of simultaneous execution (concurrent or parallel) wherein at least one thread cannot make progress
- in a modern scheduler, concurrency and parallel are absolutely going on at the same time
- loss of liveness means the thread is stuck, it's ALWAYS a problem, it's ALWAYS a bug, and it's usually the fault of the programmer
- they are very difficult to debug compared to normal bugs
- problem arises because of subtle timing mismatches in scheduling by the kernel
- it's very difficult to recreate the conditions that cause it
- the general solution is to not mess up
- follow ideas and guidelines and guardrails, and you shouldn't have problems
- we talked about three kinds of loss of liveness: deadlock, priority inversion, and livelock (ALL THREE WILL BE ON THE FINAL)
- for most of our discussions, we will focus on deadlock since it's the one most people trip over

-Priority inversion:
-From Wikipedia: "priority inversion is a scenario in scheduling in which a high-priority task is indirectly superseded by a lower-priority task effectively inverting the assigned priorities of the tasks. This violates the priority model that high-priority tasks can only be prevented from running by higher-priority tasks."
-scenario: a higher priority process (let's call it H) is blocked from accessing a shared resource because a lower priority process (let's call it L) locked out higher priority process H once it entered its critical section, thus a lower priority process is running and priorities are inverted

-Livelock:
-A livelock is a special case of resource starvation
-A livelock is when two threads are constantly changing with regards to one another without ever progressing further in reaction to one another
-The difference between a livelock and a deadlock is that a deadlock is when one thread blocks another thread by locking the other out, whereas in a livelock, both threads are running (they're not blocking) but neither can do anything to get around the other because they both keep responding to one another
-Analogy: "A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time."
-Image depicting a livelock:



-Spinlock:

-This is literally just a type of locking that looks like this: "while (*lock_variable) { }"

-This is a "busy wait" as we're using CPU while waiting for the lock to be retrieved

-Deadlock: two of more threads are waiting for an event that can only be caused by one of the waiting threads

-Analogy: two people (each person is a process) are trying to go through a door and one person needs to tokens to exit the room (terminate); a process cannot exit if it does not have both tokens

-Deadlocks can arise if four conditions hold true simultaneously in a system:

1. Mutual exclusion - at least one resource must be held in a nonsharable mode, this causes problems because of race conditions (simultaneous access of a writeable resource), a critical section is a place where we have to enforce mutual exclusion
2. Hold and wait - a thread can hold a resource without having all the resources; the waiter and arbiter solution prevents "hold and wait" from existing and thus prevents dreadlocks
3. No preemption - no thread can take another thread's stuff (analogy: no thread can "mug" another thread)
4. Circular wait - process A is waiting on something that process B has; and process B is waiting on something process A has

-Resource-Allocation Graph - a data structure (directed graph, all edges are directed) used to keep track of what processes/threads exist, what mutual exclusion resources exist; this would be too costly for the scheduler to have to search through

-There are two kinds of nodes in this data structure: line of execution (process or thread) is a circle node, resource (mutual exclusion resource, i.e. mutex) is a square node

-A box with a single dot (edge) is a unique resource

-An edge going from a circle node (process/thread) to a square node (resource) is a request (NOTE: edges never go from circle node to circle node or square node to square node; a thread cannot own another thread and a resource cannot own another resource)

-A graph that no cycle can have no deadlock; if there is a cycle, that means you are "unsafe" (in this sense, it means a deadlock COULD happen, does mean it will)

-**ON EXAM:** look at resource allocation graph and determine if there's a deadlock

-In real life, we would use an adjacency matrix to represent the resource allocation graph and determine if there were cycles

-Post example here…

3/20

Wednesday, March 20, 2024 8:11 AM

- -Three ways of deadlock avoidance:
    - -???
    - -Deadlock detection/prediction
    - -Deadlock breaking

-Deadlocks are like race conditions, they don't always show up

- -We can deal with the deadlock problem in one of three ways
    - -Ignore the problem altogether and pretend that deadlocks never occur in the system
        - -An example of deadlocks is when your router freezes up (usually because of poorly written code) so you turn it off and on again
    - -???

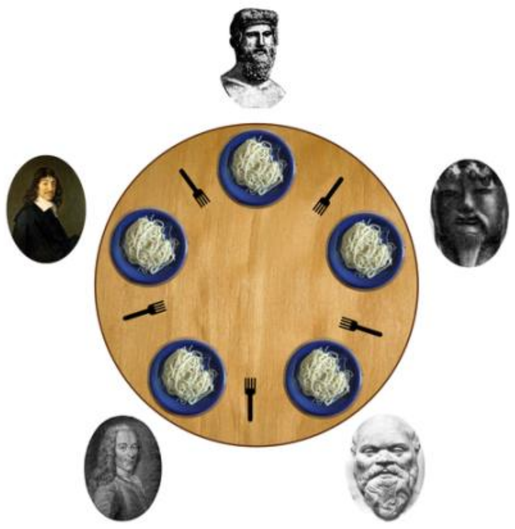
- -Both solutions (algorithms) break one of the four deadlock properties:
    - -Can't go after mutexes because we need to deal with race conditions and we can't attack race conditions (can't go after mutual exclusion and can't go after preemption)
    - -The arbiter solution / the waiter solution
        - -Attacks the wait and hold property
        - -Only the arbiter can give a thread its deadlock token
        - -Breaks hold and wait because it becomes impossible for you to forever hold a resource and not get the other one
        - -You can implement the arbiter solution with 1 mutex
        - -This is inefficient if the threads want the deadlock tokens back very often (high contention; IO bound)
        - -Any problem in engineering, "you can have it fast, you can have it cheap, or you can have it good: choose any two"
        - -The arbiter becomes a "bottleneck" and has "less contention"
    - -The resource hierarchy solution
        - -Attacks the cycle property
        - -If two threads want a resource, the thread with the highest ordering (priority) gets the resource first

3/22

Friday, March 22, 2024 8:00 AM

-Dining with Philosophers Problem (analogy)

- -Philosophers are processes/threads
- -Philosophers (processes/threads) can be in one of two states: thinking state or eating state
- -Each philosopher has a bowl of spaghetti
- -Each philosopher has one fork (deadlock token or shared resource; this is a mutex)
- -But philosophers can only eat spaghetti if they have two forks
- -Arbiter/waiter solution: an arbiter (AKA waiter) comes around and gives each philosopher a fork individually if they want one
- -Resource hierarchy solution: the philosophers ask for resources in some ordering (priority) and whichever philosopher has a more important ordering (priority) is the one who gets the resource first -- there's no arbiter in this solution; you can use the corresponding chopstick (shared mutex resource / deadlock token) to represent some ordering number



3/25

Sunday, March 24, 2024 8:22 PM

-Wednesday - 8 AM, finals week, in-person but online (via Canvas), we'll get a sample final exam (it's a superset of the final exam, some questions might be the same but lots will be rewritten), NOT open note but you get Professor Gallagher's own pre-written notes (1 or 2 pages, things that are useful for the exam but you don't need to memorize) and we should know what these open notes look like, few people who study the sample final tend to do well, the final will challenge you (half of the people will walk out of here not feeling great)

3/27

Wednesday, March 27, 2024 8:07 AM

-Frame - a fixed sized chunk of **physical** memory, referred to by **frame number** by the OS

  -Frame table - an OS data structure that keeps track of the properties of each frame (says which frames are in use, the kernel would find an open, not in-use frame table if it needs one)

-Page - a fixed sized chunk of **logical** memory, referred to by **page number** by the process

  -Page table - an OS data structure associated with each process that keeps track of what *frames* the process owns

-Page size == frame size (ALWAYS)

-Page table size is often variable (processes have different memory size)

-Frame table size is determined by the size of physical memory

-Processes do NOT access memory directly, they make reference to "virtual" addresses that go from 0 to some limit where the contents of the page table are used to translate virtual addresses into physical addresses

Example:

-A typical frame size for a Linux machine is 4K (4096 bytes of memory). For a machine with 16 GB of memory, how many frames are there?

  -Total memory / size of frame = 16 GB / 4K = 2^34 / 2^12 = 2^(34 - 12) = 2^22 = 4 MB of pages

-Code running in user mode is looking at virtual memory addresses, running the CPU in God mode means you're looking at physical memory addresses

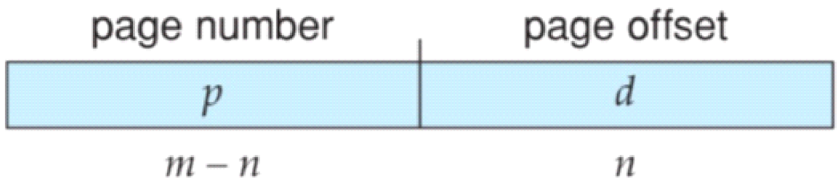
-Logical address:

  -Ranges from 0 to some max value

  -Logical addresses are broken into two fields

  -If the frame (page) size is 2^n and the address has m bits, then the first (m-n) bits is the page number; the remaining n bits is the displacement *inside* the page where the specific word lives

    -p is the page number and is represented in (m-n) bits, d is the page displacement (offset) and is represented in n bits



3/29

Friday, March 29, 2024 8:01 AM

-It's possible to have more physical memory than you do pages in logical memory

-Example

- -The PDP-11 had a 16-bit address space and a page size of 8KB. Compute how many frames were available and the maximum size of a page table.
    - -2^16 = 64KB
    - -2^13 = 8KB
    - -Let m = 16 and n = 13
    - -Page number (p) = m - n = 16 - 13 = 3
    - -Page offset (d) = n = 13

4/10

Tuesday, April 9, 2024 10:05 PM

> page table review

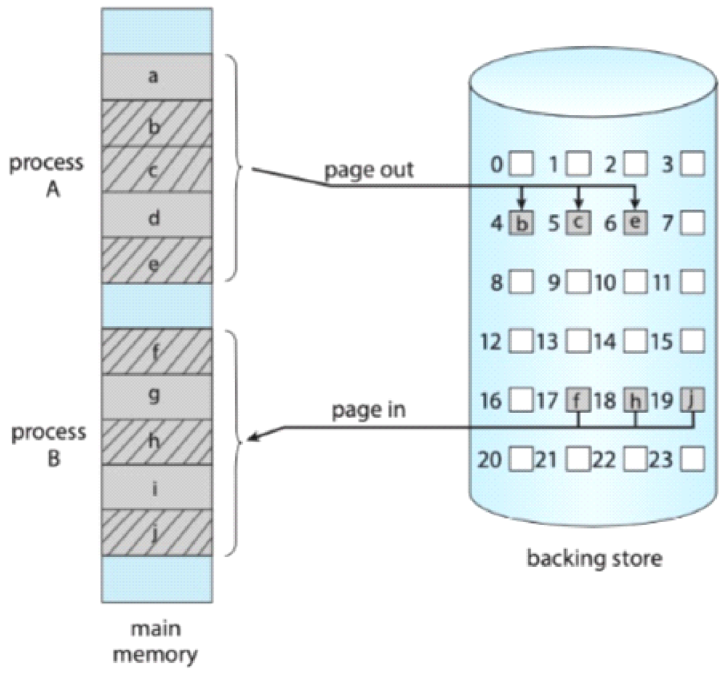
- how many page tables are there? as many as there are processes
    - -Each process gets its own page table (a page table is a logical unit of memory that points to some physical memory)
- quintessential problem is fight between size of implementation and speed of access
    - -how big is the page table vs. how fast can you access the page table
- why are you worried about speed? because memory accesses are happening CONSTANTLY
- why are you worried about size? because it lives in the kernel?
- typical page size is 4k, 2k, 8k, it changes
- how big is your page table going to be?

> frame table review

- what is a frame table? the SINGLE frame table is managed by the kernel, and it manages PHYSICAL frames of memory, it stores meta-information about the frame (like if it's being used or not, if the memory is read-only or write-only)

-What if there's too many requests for memory by many processes?
- -Analogy: you build a shed when you have too much stuff
- -You swap the pages with somewhere further out in memory (usually a disk drive or SSD)
- -The page that you choose to swap out is determined by some metric or algorithm, one metric used is putting the least recently used (LRU) page out to disk space when you're using too much RAM
    - -Some other metrics for choosing which page goes to storage: choose a random page, use a FIFO (the first page in the queue is the first page out of the queue)
    - -The perfect case is when we KNOW the exact page we won't need for the longest amount of time and send THAT page to the disk space (of course, the world isn't perfect, but we can try to forecast) -- "optimal paging strategy", all these metrics are "approximations of perfection"
    - -This can tie back into Game Theory and Nash Equilibrium where we want to find an optimal solution
    - -Just like with CPU scheduling (like in 2/21 lecture):
        - -Pareto optimality, we can't optimize two things to maximum at the same time: https://en.wikipedia.org/wiki/Pareto_efficiency
        - -"You can have it good, you can have it cheap, or you can have it fast: choose any two"
        - -"The optimist says the glass is half full, the pessimist says the glass is half empty, the engineer says the glass is twice as large as it needs to be"

Paging (Swapping with Paging)


process A
page out
process B
page in
main memory
backing store
a
b
c
d
e
f
g
h
i
j
0 1 2 3
4 b 5 c 6 e 7
8 9 10 11
12 13 14 15
16 17 f 18 h 19 j
20 21 22 23