DESCRIPTION OF IMPLEMENTATION

---

1. Condition Variable

A Condition Variable is defined as a struct in kernel/condvar.h. It has just one field: int that recognizes it.

The implementation (in condvar.c) contains 3 functions, as described below :

- cond_wait : Calls the condsleep() function. The condsleep() function, apart from doing everything the sleep() function does, goes to sleep on the condition variable with which the function was called.
- cond_signal : Calls the wakeupone() function. The wakeupone() function, instead of waking up all the processes waiting on the channel (i.e. what wakeup() does) of this condition variable, only wakes up one of the processes. For this, while looping through all the processes, as soon as we get the first process that was sleeping, we set *found* (local variable) to 1 and break out of the loop using it. Hence, we only set one process state as RUNNABLE.
- cond_broadcast : Calls the wakeup() function. The wakeup() function wakes up all the processes waiting on the channel of this condition variable. While looping through all the processes, it checks if the state of the current process in the list is SLEEPING and that it is waiting on the passed condition variable channel. For all such processes, it updates the state to RUNNABLE.

---

2. Semaphores

A Semaphore is defined as a struct in kernel/semaphore.h. It has 3 fields : The value of the semaphore, a lock (we use a sleeplock in our implementation), and the condition variable.

The implementation of semaphore (in semaphore.c) contains 3 functions, as described below :

- sem_init : Initialises the value of the counting semaphore to the given value. A sleeplock is used in the implementation, which is also initialised here.
- sem_wait : A process can only go into the critical section if the semaphore holds a positive value. If this is not the case, the process waits until some other processes post to(increments) the semaphore. This condition is implemented using the cond_wait function. If the value is 0, the process is sent to sleep. Whenever some process posts to the semaphore, the sleeping process is signalled and moved to the runnable state. It decrements

the value of semaphore (which is atomic since the sleeplock is acquired by the process), and finally releases the lock.

- sem_post : To keep the update to the semaphore atomic, sleeplock is acquired first. After incrementing the semaphore value, the cond_signal call wakes up one of the processes from the waiting queue. Finally, the lock is released.

---

3. barrier()

struct barrier_elem defines the structure of a barrier (in barrier.h). It consists of 4 elements:

- Integer Count: It keeps track of the number of processes that have reached and are waiting at the barrier. When a process reaches a barrier(calls barrier()), we increment the count indicating that another process has reached the barrier.
- Condition Variable (cv): When a process reaches a barrier, if it is the n^{th} process, we broadcast a signal to all the processes to wakeup and continue execution (cond_broadcast). However if it isn't the n^{th} process, we put it to sleep using the condition variable(cond_wait).
- Sleeplock (lock): Since incrementing the count and assessing the condition variable are critical sections, we restrict their access to only one process at a time using a sleeplock. Not doing so can lead to deadlocks.
- Integer IsFree: It is a flag telling us if the particular barrier in the barrier array is free or not. We can only assign a free barrier to a process using 'barrier_alloc'

---

4. barrier_alloc()

Since we've declared a barrier array(of size 10) consisting of *struct barrier_elem* we have to initialise the elements before accessing it. We maintain a variable *barrierInitialized* , the first time this function is called, we initialise the array and set *barrierInitialized=1.*
We loop through the elements of the *barrier_array* looking for an element which has *isFree=1*. We set the variable *isFree=0* and then return the index(id) of this barrier, indicating that this barrier(element) has been assigned to the program.

---

5. barrier_free()

It just frees the barrier, i.e. it sets *itFree=1.* The barrier is now free and can be allotted to some other program(by calling barrier_alloc).

---

6. **buffer_cond_init()**

This is where we initialise all the locks, namely: *lock_delete* (so that multiple consumers don't consume simultaneously), *lock_insert* (so that multiple producers don't insert buffer simultaneously), *lock_print* (to print properly) and the corresponding 20 buffer locks (so that a buffer isn't concurrently accessed by a consumer and a producer). Along with this, we also set the global variables *tail* (indicates where producer adds next) and *head* (indicates where the consumer deletes from next) to zero.

---

7. **cond_produce()**

We atomically and circularly (i.e. modulo BUFFER_SIZE) increment *tail* (to indicate where the next production will be) and store the current index where insertion is to happen in *index* (local variable). We wait on the *deleted* field (condition variable) of the buffer if the buffer at *index* is full (consumer has not yet deleted the last inserted buffer). This is woken up by the corresponding signal from cond_consume() function. If the buffer is empty or after it is signalled (i.e. it is out of the loop), we put the *value* in the buffer, mark it as full and signal for any condition variables waiting on the *inserted* condition variable field of the current buffer in the cond_consume() function.

---

8. **cond_consume()**

We atomically and circularly (i.e. modulo BUFFER_SIZE) increment *head* (to indicate where the next deletion will be) and store the current index where deletion is to happen in *index* (local variable). We wait on the *inserted* field (condition variable) of the buffer if the buffer at index is empty (producer has not yet added it). This is woken up by the corresponding signal from cond_produce() function. If the buffer is not empty or after it is signalled, we consume it (store in a local variable and mark it as empty) and signal for any condition variables waiting on the *deleted* condition variable field of the current buffer in the cond_produce() function. We also print the value consumed atomically and return it in the end.

---

9. **buffer_sem_init()**

- We use 4 counting semaphores in our implementation of the bounded buffer problem. The semaphores 'empty' and 'full' tell us how many subsequent productions and consumptions can occur in the buffer. The semaphores 'pro' and 'con' are used to ensure that the productions and consumptions in a buffer location are atomic.

- Next, we initialise the values in the buffer array to a default value -1. In the implementation, we have taken the size of the buffer array to be 20.
- We also initialise the two pointers *nextp* and *nextc* to 0 which tell us the location in which the next production/consumption will occur in the buffer array.

---

10. sem_produce()

- The productions in the buffer are done in a cyclic order. Since we need to ensure the atomicity of writes to the buffer (so that productions are not overwritten by competing producers), we assign value 1 to the semaphore 'pro' and call sem_wait() on both 'pro' and 'empty'.
- If 'empty' has value 0, this means that the buffer has no empty slots. In this case, the producer goes to sleep(through the cond_wait function in the implementation of sem_wait [implemented in semaphore.c]) until there are empty slots in the buffer.
- Otherwise, we can go into the critical section and write to the buffer. 'pro' ensures that the write is atomic among producers.
- Finally, once the value is written, we call sem_post on 'pro' so that other producers can enter into the critical section. sem_post is also called on 'full' since there is one more element available to be consumed.

---

11. sem_consume()

- This function follows a similar approach as sem_produce(). We use the 'con' variable to ensure atomicity in consumptions, as we did for productions using 'pro'.
- We call sem_wait on con to ensure there are elements available to be consumed in the buffer.
- Subsequently, consumer enters the critical section, consumes the value, and posts to 'empty', since there is one more available slot in the buffer. **NOTE :** The consumptions are implemented in a FIFO order.
- Finally, we return the consumed value.

ANALYSIS OF IMPLEMENTATIONS

INPUT: 40 10 5

Condprodconstest :

```
$ condprodconstest 40 10 5
Start time: 988

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 56 55 57 58 60 59 61 62 63 65 64 66 67 68 69 80 120 2
80 320 70 360 71 72 73 74 75 76 81 77 82 78 83 200 160 240 79 84 241 281 321 85 361 86 122 87 123 88 89
124 125 90 126 91 127 162 202 242 282 322 362 128 92 93 129 94 130 95 131 121 161 201 96 132 283 323 363
 97 133 98 134 99 135 100 136 101 137 102 138 203 163 243 139 103 324 364 284 104 140 105 141 106 164 10
7 204 108 244 109 145 143 144 142 245 110 325 285 365 111 112 147 148 149 113 150 114 115 151 116 152 16
5 146 205 206 117 286 326 366 154 118 119 167 155 166 168 169 156 170 157 171 158 246 153 159 287 172 36
7 173 208 174 209 175 210 211 177 212 178 213 214 179 215 248 216 288 176 328 368 180 181 182 183 184 18
5 186 217 249 247 289 329 369 188 187 189 191 190 192 194 193 207 195 197 196 198 327 218 290 199 330 37
0 219 220 251 221 252 222 223 253 254 224 255 225 226 256 250 257 227 331 371 228 229 258 230 259 231 26
0 232 261 233 262 234 263 235 264 291 236 332 265 372 237 266 238 267 239 268 269 293 270 294 271 333 27
2 334 273 292 335 274 295 275 336 373 276 297 277 298 278 299 279 296 300 301 337 338 302 339 340 303 34
1 374 304 305 342 343 344 345 346 347 348 386 349 375 377 376 378 380 379 381 382 383 385 306 387 388 38
9 390 391 350 384 392 351 393 307 308 309 310 311 312 313 314 315 316 317 318 319 352 353 354 355 356 39
4 357 358 359 395 396 397 398 399

End time: 991
```

Time taken = 3 units

Semprodconstest :

```
$ semprodconstest 40 10 5
Start time: 114

0 1 40 80 41 81 42 82 43 2 3 4 5 6 7 8 9 10 11 12 13 14 44 83 45 84 46 15 16 17 18 47 48 240 241 19 280
20 281 21 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 360 49 361 50 362 51 363 52 36
4 22 23 24 25 26 27 28 85 120 86 121 160 122 29 53 161 30 31 32 33 34 87 35 54 55 36 37 88 123 89 162 90
 124 125 38 126 56 39 57 58 163 59 91 92 93 94 95 96 97 98 99 100 101 127 128 129 130 320 60 61 62 63 64
 65 66 67 68 69 70 71 72 73 74 102 131 103 132 104 133 134 135 136 137 138 139 140 141 142 75 143 105 76
 77 106 78 79 107 108 109 110 111 112 113 114 115 116 117 144 164 145 165 146 166 147 167 118 168 119 16
9 148 200 149 170 171 172 173 321 322 323 150 151 152 153 154 155 156 157 158 159 174 175 176 177 178 17
9 180 181 182 183 184 185 186 187 188 201 202 203 204 189 190 191 192 193 194 195 196 197 198 199 205 20
6 207 208 258 282 324 283 325 284 285 286 287 326 327 328 329 330 331 209 259 260 210 211 212 213 214 21
5 216 217 218 261 288 262 289 263 290 264 291 219 220 221 222 223 224 332 225 333 226 334 227 265 266 26
7 228 229 230 231 232 233 234 235 268 269 270 271 292 236 237 238 239 272 273 274 275 276 277 278 279 29
3 335 294 336 295 337 365 296 297 298 299 338 300 366 301 367 302 368 303 369 304 339 340 341 342 305 30
6 307 308 309 310 311 312 313 314 315 316 317 318 319 343 344 345 346 347 348 349 350 351 352 353 354 37
0 355 356 357 358 359 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 39
1 392 393 394 395 396 397 398 399

End time: 119
```

Time taken = 5 units

INPUT : 100 5 2

Condprodconstest :

```
$ condprodconstest 100 5 2
Start time: 449

0 1 2 3 4 5 6 7 8 100 9 101 102 103 10 11 105 12 106 13 107 14 108 15 109 104 110 16 17 111 200 18 300 4
00 19 112 113 114 115 116 117 118 119 120 121 20 122 132 29 21 22 23 24 28 25 26 123 124 125 126 127 128
 129 131 130 27 201 133 401 134 142 202 402 31 32 33 135 34 136 35 30 137 138 36 139 140 37 141 301 38 1
43 203 39 403 40 204 144 302 303 304 305 404 145 205 146 206 147 207 148 208 149 209 150 210 211 151 212
 306 41 152 42 153 43 44 154 45 155 46 156 47 157 48 158 49 159 214 50 307 213 51 160 52 161 53 162 54 1
63 164 55 56 165 57 166 58 59 167 216 215 308 60 61 62 168 169 170 63 171 64 173 172 65 174 66 175 67 18
6 217 176 68 309 69 70 71 177 178 72 179 180 73 181 182 74 184 183 185 405 75 91 92 406 187 76 78 79 80
81 82 83 84 85 77 86 87 88 89 90 188 218 189 310 93 407 190 94 191 95 96 192 193 97 194 195 98 99 196 19
7 219 198 220 311 408 199 221 222 223 312 224 313 225 314 226 315 227 316 228 317 229 230 318 409 231 23
3 319 232 320 234 322 235 323 236 324 237 325 238 326 239 321 240 327 410 241 242 243 245 244 246 328 32
9 247 330 248 331 249 250 251 332 333 252 411 334 335 253 336 254 337 255 338 256 339 257 258 340 341 25
9 342 343 260 261 344 412 262 263 345 264 265 347 266 348 267 349 268 350 269 346 351 270 352 271 353 27
2 413 273 275 276 277 278 279 280 281 282 283 284 274 354 285 286 287 288 289 290 291 292 293 294 295 29
6 297 414 298 355 415 356 416 299 417 418 419 420 421 422 423 424 425 426 427 428 357 358 359 360 361 36
2 429 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 431 432 378 433 434 430 435 436 437 43
8 439 440 441 442 443 444 454 445 446 447 448 449 450 452 451 453 455 456 457 458 459 460 461 462 463 46
4 379 380 381 382 383 384 385 386 387 465 388 389 390 391 392 393 394 395 396 397 398 466 399 467 468 46
9 470 471 472 473 474 475 476 478 479 480 481 482 483 484 485 486 477 488 487 489 490 491 492 493 494 49
5 496 497 498 499

End time: 454
```

Time taken = 5 units

Semprodconstest :

```
$ semprodconstest 100 5 2
Start time: 47

0 1 2 3 4 5 6 7 8 9 100 200 101 201 102 202 103 10 11 12 13 14 15 104 16 105 203 106 204 300 400 401 402
 403 404 17 405 406 18 107 19 108 20 109 205 301 407 408 409 410 411 412 413 414 415 416 417 418 419 420
 421 422 21 423 424 425 426 427 428 429 22 430 206 431 207 432 23 110 111 112 113 24 114 433 115 208 209
 210 302 303 304 305 306 307 434 435 436 25 26 116 27 117 28 118 29 119 30 308 31 309 32 310 33 437 120
211 212 213 214 215 216 217 218 219 220 221 222 34 223 35 36 37 38 39 40 121 122 123 41 42 124 224 125 2
25 126 127 43 128 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 226 227 228 229 230 231 232 233 234
 235 236 237 238 61 62 63 64 65 66 67 68 69 70 71 72 73 129 74 75 76 77 78 79 80 81 82 83 84 130 131 132
 133 134 135 136 137 138 139 140 141 142 85 86 87 88 143 239 144 89 90 91 92 93 94 95 96 97 98 99 145 14
6 240 241 242 243 244 245 147 148 149 150 151 152 153 154 155 156 157 158 246 311 247 312 248 159 160 16
1 162 163 164 249 313 314 315 316 317 318 319 320 321 322 323 324 325 326 438 439 165 250 251 252 253 25
4 166 255 440 256 441 257 258 259 260 261 167 262 168 263 169 327 170 264 265 266 267 171 172 442 173 44
3 174 268 269 270 175 271 272 176 273 274 177 275 178 276 179 328 180 181 182 183 277 184 185 186 187 27
8 188 279 189 190 191 280 329 330 444 192 445 331 446 193 281 282 283 284 285 286 287 194 288 289 290 29
1 292 293 294 295 332 333 195 196 197 296 198 199 297 298 299 334 447 335 448 336 449 337 450 338 339 34
0 341 342 451 452 453 454 455 456 343 457 344 458 345 346 347 348 349 350 351 352 353 354 355 459 460 46
1 462 463 464 465 466 467 468 469 470 471 472 473 356 474 357 475 358 476 359 477 360 361 362 363 364 36
5 366 367 368 369 370 371 372 373 374 375 376 478 377 479 378 379 380 381 382 480 383 481 384 482 385 38
6 387 388 389 390 391 392 393 394 395 396 397 398 483 399 484 485 486 487 488 489 490 491 492 493 494 49
5 496 497 498 499

End time: 54
```

Time taken = 7 units

INPUT: 30 8 8

Condprodconstest :

```
$ condprodconstest 30 8 8
Start time: 136

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 20 22 23 31 24 25 33 26 34 27 28 29 35 36 60 61 19 21 30 32 37 39 40 41 42 43 44 45 46 47 48 53 150 49
50 51 52 210 120 38 180 90 211 55 63 121 64 57 65 58 66 59 67 68 91 69 62 181 54 151 56 212 71 93 72 92 73 123 74 124 75 125 76 126 77 70 182 122 152
127 79 80 81 82 83 84 85 86 87 88 89 128 95 96 98 97 78 183 94 213 153 214 101 130 102 131 103 132 129 104 133 105 184 134 106 135 107 136 99 154 100
137 185 109 138 108 110 139 111 140 112 141 113 142 114 143 115 144 145 155 215 146 116 216 117 118 156 148 119 186 149 157 187 158 188 159 189 160 19
0 161 147 217 163 192 191 162 193 218 194 219 220 195 221 222 196 223 224 197 225 164 165 166 199 167 200 198 201 227 202 228 203 204 229 230 205 231
226 232 207 206 168 169 208 209 170 171 233 234 235 236 237 238 239 173 174 175 176 172 177 178 179

End time: 141
$
```

Time taken: 5 units

Semprodconstest:

```
$ semprodconstest 30 8 8
Start time: 49

0 1 2 3 30 31 90 32 91 60 61 62 63 64 65 66 67 68 69 70 92 93 94 95 150 4 33 71 34 72 35 73 5 6 36 37 38 39 40 41 42 74 43 75 44 7 45 8 46 9 47 10 96
151 152 153 154 155 156 157 11 76 12 77 13 97 14 98 48 78 79 80 15 81 82 83 84 16 17 85 18 86 19 87 20 88 49 21 22 23 24 25 26 27 28 180 181 182 183 1
84 185 186 187 188 189 190 191 192 29 50 89 99 120 51 121 52 122 100 53 101 54 102 55 103 56 104 57 58 59 105 106 123 107 124 108 125 158 193 210 211
212 213 214 109 215 110 111 112 126 113 114 115 116 117 118 119 127 128 129 130 131 132 194 133 195 134 196 135 197 136 159 160 161 162 163 137 164 16
5 166 167 168 169 170 171 172 138 198 139 199 140 200 141 201 173 174 142 175 143 176 144 177 145 178 146 179 147 216 148 217 149 218 202 219 203 204
205 206 207 208 220 221 222 223 224 225 209 226 227 228 229 230 231 232 233 234 235 236 237 238 239

End time: 57
```

Time taken: 8 units

- The implementation using condition variables is faster for the bounded buffer problem. This is because of the added concurrency present in this version.
- In the implementation using condition variables, the cond_wait function is called on each slot of the buffer. This means that multiple productions and consumptions may occur on the buffer array concurrently. The only restriction being that only 1 producer/consumer can stay at a time on a buffer spot. Once the producer has produced, no other producer produces on it, i.e. it is not over-written until a consumer consumes it. Similarly, once a consumer has consumed, no other consumer consumes it, i.e. it is not consumed again until another producer has produced on it.
- In the implementation of the bounded buffer using semaphores, to avoid problems like productions being overwritten/the same element being consumed twice, we made the productions and consumptions atomic. However, this means that the producers and consumers can no longer be concurrent among themselves. The only form of concurrency we have is if one process produces to the buffer while some other process consumes.